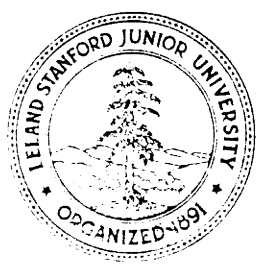
SOME THOUGHTS ON PROVING CLEAN TERMINATION OF PROGRAMS

BY

Richard L. Sites

STAN-CS-74-417
MAY 1974

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY



Some Thoughts on Proving Clean Termination of Programs

by Richard L. Sites

Abstract

Proof of clean termination is a useful sub-goal in the process of proving that a program is totally correct. Clean termination means that the program terminates (no infinite loops) and that it does so normally, without any execution-time semantic errors (integer overflow, use of undefined variables, subscript out of range, etc.). In contrast to proofs of correctness, proof of clean termination requires no extensive annotation of a program by a human user, but the proof says nothing about the results calculated by the program, just that whatever it does, it terminates cleanly. Two example proofs are given, of previously published programs: TREESORT3 by Robert Floyd, and SELECT by Ronald L. Rives-t and Robert Floyd.

This work was supported in part by the Fannie and John Hertz Foundation, by the National Science Foundation and by IBM Corporation. Reproduction in whole or in part is permitted for any purpose of the United States Government.

CERTIFICATION OF ALGORITHM 245[M1]

TREESORT3 [Robert W. Floyd, Comm. ACM 7 (Dec. 1964), 701]:

PROOF OF CLEAN TERMINATION -- A NEW KIND OF PARTIAL CERTIFICATION

Richard L. Sites
Computer Science Department
Stanford University
Stanford, California 94305

Abstract

The certification of a program can include a proof that the program always terminates cleanly, i.e., that as it runs on a real machine, it generates no semantic errors and it encounters no infinite loops. As an illustrative example, a previously certified algorithm, TREESORT3, is examined and a hidden restriction is exposed which prevents it from running properly on some machines.

Keywords and Phrases: proof of termination, debugging, certification, sorting, proof of correctness.

CR Categories: 4.42, 4.49, 5.24, 5.31

This certification differs from London's certification [2] in two important respects: (1) it deals explicitly with running the algorithm on a real machine which has restrictions on the validity of arithmetic operations (roundoff error, overflow); (2) it deals only with proving that the algorithm terminates cleanly, without examining what it accomplishes (i.e., without proving that it sorts an array).

The need for such a certification follows from the fact that TREESORT3 will actually fail in realistic situations, although it has been "rigorously proved correct". This flaw was noted in London's reply to Redish [3].

Proving that an algorithm terminates cleanly means proving that as it runs on a real machine it generates no semantic errors and that it encounters no infinite loops. A semantic error is produced by attempting any operation which the language specifies to be illegal or undefined, or any operation which violates a restriction of a particular implementation of the language. Many implementations fail to detect all semantic errors at run-time; this produces meaningless results and is one of the tragedies of our profession. Common semantic errors include arithmetic overflow, underflow, division by zero, subscript out of range, case or switch expression out of range, use of uninitialized variables, and use of a null pointer.

In the discussion below, it is assumed that the algorithm will run on an ALGOL 60 machine with the following properties:

1. Integer overflow. The binary operations i+j , i-j , i×j , i÷j , and i/j give the mathematically correct result if and only if i and j have defined values and the result is in the range I_{min} to I_{max} inclusive; otherwise a semantic error occurs. It is

assumed that $I_{min} < 0$ and $I_{max} > 0$. Division by zero produces a result outside of the range I_{min} to I_{max} ; i.e., a semantic error occurs.

2. No assignment of **uninitialized values.** The operation i :=j will assign the value of j to i if and only if j has a defined value; if j is uninitialized then a semantic error occurs. It is possible to write algorithms which violate this restriction and still give meaningful results, but more often violation of this condition indicates an error which is best caught as soon as possible.

3. Subscript range checking. If A is an array with bounds $[A_\ell : A_u]$ then in all references to A[i] , it must be true that i is defined and $A_\ell \leq i \leq A_u$.

4. Mathematically correct **comparison.** The relation i < j always produces the proper value true or false, even in cases where j-i would produce an overflow. On a machine which has no compare instruction, such as the CDC 6600, this property is not true; thus, algorithms which are certified to execute and terminate cleanly on the 6600 must be transformed so that every comparison is done as a subtraction and a sign test, then all the subtractions checked for overflow/underflow. Two representations of zero are allowed if the implementation gives identical results for each.

5. Representable constants. Each integer constant must be between I_{min} **and** I_{max} inclusive.

The proof of clean termination of TREESORT3, under suitable restrictions on the parameters, is presented below in five parts: a copy of the algorithm [1], the corresponding flow graph, a listing of

the assertions about semantic errors, a listing of the assertions about loop termination, and proofs of the assertions. An appendix extends the analysis to machines like the CDC 6600.

ALGORITHM 245
TREESORT3 [M1]

Robert W. Floyd (Recd. 22 June 1964 and 17 Aug. 1964)
Computer Associates, Inc., Wakefield, Mass.

```
procedure TREESORT 3 (M,n);
   value n; array M; integer n;
comment TREESORT 3 is a major revision of TREESORT [R. W. Floyd,
   Alg. 113, Comm. ACM 5 (Aug. 1962), 434] suggested by HEAPSORT
   [J. W. J. Williams, Alg. 232, Comm. ACM 7 (June 1964),347]
   from which it differs in being an in-place sort. It is shorter and
   probably faster, requiring fewer comparisons and only one division.
   It sorts the array M[1:n] , requiring no more than 2 x(2 ↑ p-2) x(p-1) ,
   or approximately 2 xn x (log2(n)-1) , comparisons and half as many
   exchanges in the worst case to sort n = 2 ↑p-1 items. The algorithm
   is most easily followed if M is thought of as a tree, with M[j ÷2]
   the father of M[j] for 1 < j ≤ n ;
begin

   procedure exchange (x,y); real x,y;
      begin real t; t :=x; x :=y; y :=t
      end exchange;

   procedure siftup (i,n); value i,n; integer i,n;
   comment M[i] is moved upward in the subtree of M[1:n] of which
      it is the root;
   begin real copy; integer j;
         copy :=M[i];
      loop: j :=2 xi;
         if j ≤ n then
         begin if j < n then
               begin if M[j+1] > M[j] then j :=j+1 end;
            if M[j] > copy then
               begin M[i] :=M[j]; i :=j; go to loop end
         end;
         M[i] :=copy
   end siftup;
```

```
    integer i;
    for i := n ÷ 2 step -1 until 2 do siftup (i,n);
    for i :=n step -1 until 2 do
    begin siftup (1,i);
       comment   M[÷2] ≥ M[j] for 1 < j ≤ i;
       exchange (M[1],M[i]);
       comment M[i:n] is fully sorted;
    end
end TREESORT 3
```

The flow graphs are shown in Figures 1, 2, and 3. For reference purposes, the arcs are numbered, and the nodes are lettered. The symbol ω is used to represent the value "undefined". Following the block structure rules of ALGOL 60, all local variables are set to ω at entry to the block.

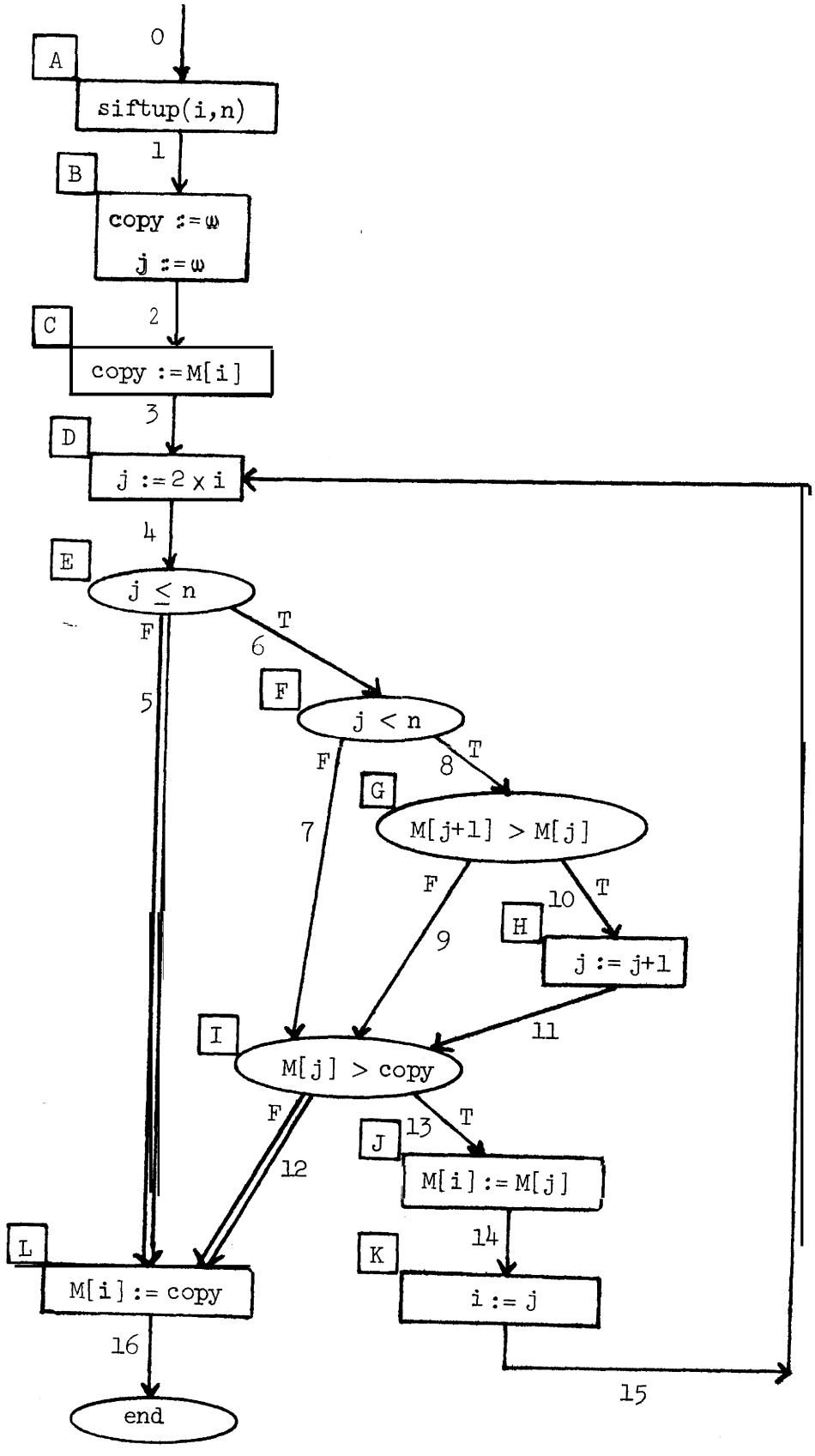


Figure 1. Flow graph of siftup. Double lines indicate loop exits.

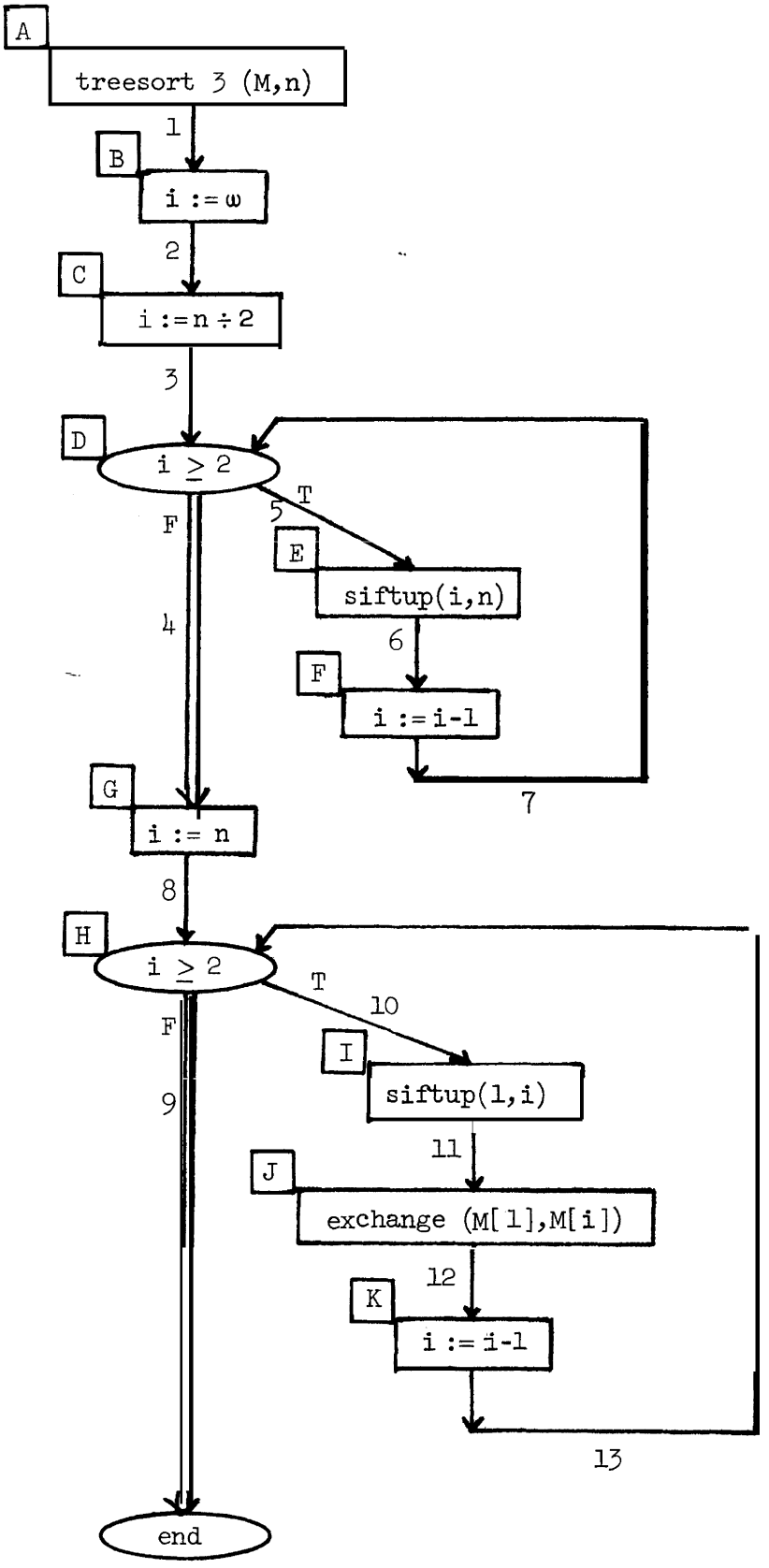


Figure 2. Flow graph of treesort 3.

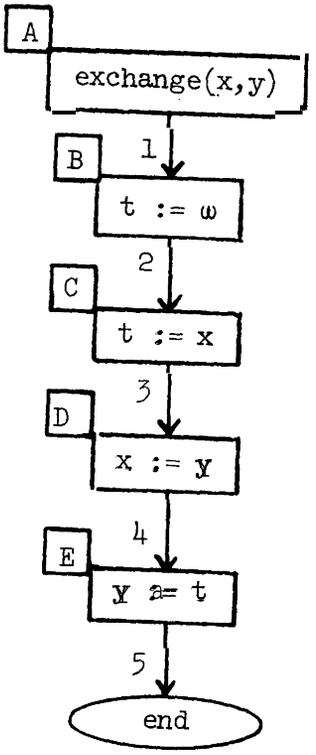


Figure 3. Flow graph of exchange.

Assertions About Semantic Errors

Assertions are generated locally and mechanically, based on the operators in a node. The assertions about a node are attached to all arcs which enter that node. The mechanical style of assertion generation and proof is intended to mimic a machine-generated certification of clean termination.

Assertions for siftup

Node	Assertion	Reason generated
C	$i \neq \omega$	Cannot use uninitialized variable.
	$M_\ell \leq i \leq M_u$	Subscript range. M_ℓ and M_u are the lower and upper bounds assumed for array M .
	$M[i] \neq \omega$	Cannot use uninitialized variable.
D	$i \neq \omega$	Cannot use uninitialized variable.
	$I_{min} \leq 2 \leq I_{max}$	Constants must be in the representable range.
	$I_{min} \leq 2 \times i \leq I_{max}$	Integer overflow.
E	$j \neq \omega$	
	$n \neq \omega$	
F	$j \neq \omega$	
	$n \neq \omega$	

Node	Assertion	Reason generated
G	$j \neq \omega$	
	$I_{min} \leq 1 \leq I_{max}$	
	$I_{min} \leq j+1 \leq I_{max}$	
	$M_\ell \leq j+1 \leq M_u$	
	$M[j+1] \neq \omega$	
	$j \neq \omega$	
	$M_\ell \leq j \leq M_u$	
	$M[j] \neq \omega$	

To keep this presentation more readable, the following trivial assertions will be elided:

1. $I_{min} \leq$ constant $\leq I_{max}$ — Assertions describing the largest and smallest constants in each procedure will be added at the end.

2. $v \neq \omega$ — Dropped when there is some other assertion about v at the same node, i.e., any assertion about the value of variable v implies the additional assertion $v \neq \omega$.

3. Any true expression involving only constants.

Node	Assertion	Reason generated
H	$I_{min} \leq j+1 \leq I_{max}$	
I	$M_\ell \leq j \leq M_u$	
	$M[j] \neq \omega$	
	copy $\neq \omega$	

Node	Assertion	Reason generated
J	$M_\ell \leq i \leq M_u$ $M_\ell \leq j \leq M_u$ $M[j] \neq \omega$	
K	$j \neq \omega$	
L	$M_\ell \leq i \leq M_u$ $copy \neq \omega$	
also	$2 \leq I_{max}$	Largest constant in procedure. Smallest constant is 1 , which is greater than I_{min} by assumption that $I_{min} < 0$.

Assertions for treesort3

Node	Assertion	Reason generated
C	$n \neq \omega$	
D	$i \neq \omega$	
E	$i \neq \omega$ $n \neq \omega$	Arguments passed to value parameters must be defined at time of call.

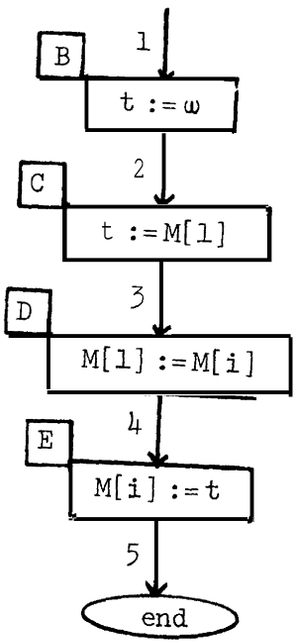
[Other assertions about the arguments to **siftup** will be inserted here after **siftup** is completely analyzed.]

F	$i \neq \omega$ $I_{min} \leq i-1 \leq I_{max}$	
G	$n \neq \omega$	
H	$i \neq \omega$	
I	$i \neq \omega$	Value parameter.
J	[Assertions relating to name parameters are all pushed into a copy of EXCHANGE associated with this particular call.]	

Node	Assertion	Reason generated
K	$i \neq \omega$	
	$I_{min} \leq i\text{-}1 \leq I_{max}$	
also	$2 \leq I_{max}$	Largest constant in procedure.

Assertions for exchange

Because it has NAME parameters, exchange must be treated in strict accordance with the copy rule. A copy of exchange is made for each call, with appropriate argument substitutions. The call at node J of treesort3 is equivalent to:



Node	Assertion	Reason generated
C	$M_\ell \leq 1 \leq M_u$	Subscript range.
	$M[1] \neq \omega$	Cannot use uninitialized variable.
D	$M_\ell \leq 1 \leq M_u$	
	$M_\ell \leq i \leq M_u$	
	$M[i] \neq \omega$	
E	$M_\ell \leq i \leq M_u$	
	$t \neq \omega$	

Assertions About Loop Termination

siftup

For any loop, asserting that it terminates is the same as asserting that there exists a $k \geq 1$ such that on the k-th iteration of the loop, one of the paths leading to an exit arc will be taken. The **siftup** procedure has just one loop, in the sense that all cycles in the flow chart pass through node D . The notation i_k means the value of i at the beginning of the k-th iteration of the loop, i.e., just before the execution of the statement j := 2 x i in node D .

The generalized loop termination assertion is:

$\exists k \geq 1$ s.t. [arc 5 taken] or
[arcs 6, 7, and 12 taken] or
[arcs 6, 8, 9, and 12 taken] or
[arcs 6, 8, 10, 11, and 12 taken] .

This expression is expanded to reflect the branches taken to reach a particular arc, using expressions in terms of values at the beginning of the k-th iteration of the loop:

$\exists k \geq 1$ s.t. $\{2 \times i_k > n_k\}$ or
$\{2 \times i_k \leq n_k$ and $2 \times i_k \geq n_k$ and $M_k[2 \times i_k] \leq copy_k\}$ or
$\{2 \times i_k \leq n_k$ and $2 \times i_k < n_k$ and
$M_k[2 \times i_k + 1] \leq M_k[2 \times i_k]$ and $M_k[2 \times i_k] \leq copy_k\}$ or
$\{2 \times i_k \leq n_k$ and $2 \times i_k < n_k$ and
$M_k[2 \times i_k + 1] > M_k[2 \times i_k]$ and $M_k[2 \times i_k + 1] \leq copy_k\}$.

In general, a loop termination assertion such as this is unprovable, but there are a few important special cases which work for many loops.

Two such cases are (1) strictly monotonic sequences of integers and (2) pointers indexing through finite linked lists. In the **siftup** loop, the fact that $n_k - j_k$ is monotonically decreasing is one key to proving loop termination.

treesort 3

first loop:

$\exists\ k \geq 1$ s.t. $i_k < 2$

second loop:

$\exists\ k \geq 1$ s.t. $i_k < 2$

exchange

No loops.

There is a shortcut to this general method of proving loop termination which completely avoids analysis of the branching structure of the loop: If any variable is monotonically increasing or decreasing inside the loop, and the loop generates no semantic errors, then it terminates! If a monotonic variable is found, then no other assertions, proofs, or loop analysis is needed. The statements which modify the monotonic variable cannot be executed an unbounded number of times without generating an overflow, so proving that those statements generate no overflow simultaneously proves that the loop terminates. This shortcut is applicable to all the loops in **treesort3** and **siftup** .

Proofs of Assertions

As indicated in the program's initial comment, all intended references to the array M involve subscripts in the range 1 to n. In case the value of n is changed in the program, we will define the variable n_0 equal to the value of n upon entry to treesort3 . All subscript range assertions will assume that $M_\ell = 1$ and $M_u = n_0$. */

Assertions for siftup (note that n and i are bound in siftup, not treesort3).

Arc	Assertion	Proof
2	$i \neq \omega$	Value parameters are always defined.
	$1 \leq i \leq n_0$	Not clear. Push back to arc 1 as an entry condition for siftup. Eventually verify that this assertion is true at each point of call.
	$M[i] \neq \omega$	Again, not clear. In fact, there is a need to assert that $\forall\ 1 \leq \ell \leq n_0$, $M[\ell] \neq \omega$ at the very beginning of treesort3 . Right now, push this assertion back to arc 1.
3	$i \neq \omega$	Used in previous node, hence i is defined or a previous assertion would be false.
	$I_{min} \leq 2 \leq I_{max}$	The assertion $I_{min} \leq 2$ is true because of the assumption that $I_{min} < 0$. Push the assertion $2 \leq I_{max}$ back to arc 1 as an entry condition for siftup .
	$I_{min} \leq 2 \times i \leq I_{max}$	Not clear. In fact, overflow will occur if $i = n_0$ and $n_0 > I_{max}/2$. We know from arc 2 that $1 < i$, so $I_{min} \leq 2 \times i$. Push the assertion $2 \times i \leq I_{max}$ back to arc 1.

*/ It is possible to develop the proof of clean termination using only M_ℓ and M_u , but doing so makes it significantly harder to prove that the siftup loop terminates when $M_a = 0$.

Analysis of loop.

Values of variables at beginning of k+1 -st iteration in terms of values at beginning of k-th iteration:

$n_{k+1} = n_k$

$copy_{k+1} = copy_k$

$i_{k+1} = 2 \times i_k$ or $2 \times i_k + 1$

$j_{k+1} = i_{k+1} = 2 \times i_k$ or $2 \times i_k + 1$

$M_{k+1} = M_k$ except $M_{k+1}[i_k] = M_k[2 \times i_k]$ or $= M_k[2 \times i_k + 1]$

$M_{k+1}[i_{k+1}] > copy_{k+1}$

$1 \leq i_{k+1} \leq n_0$

$i_{k+1} \leq n_k$

Thus n and copy are invariant in the loop; i is monotonically increasing (since $2 \times i > i$ when $i \geq 1$); various elements of M change values.

arc	Assertion	Proof
4	$j \neq \omega$	j is defined in previous node.
	$n \neq \omega$	n is defined originally as a value parameter, and is invariant within the loop.
5	$1 \leq i \leq n_0$	Still true from arcs 3 and 15.
	$copy \neq \omega$	True from node C and copy invariant in loop.
6	$j \neq \omega$	True from previous node.
	$n \neq \omega$	True from previous node.

arc	Assertion	Proof
7	$1 \leq j \leq n_0$	Not clear. On edge 7, j = n , and n is invariant within the loop, so push the assertion $1 < n \leq n_0$ to arc 1.
	$M[j] \neq \omega$	Willbetrueif $\forall\ 1 \leq \ell \leq n_0$, $M[\ell] \neq \omega$ on arc 1, since after that, an undefined value can never be assigned to an element of M in our machine model.
	copy $\neq \omega$	True from node C, and copy is invariant within the loop.

For a more readable presentation we will elide the statement and proof for all assertions which are true because they were true earlier in the program and haven't changed.

8	$I_{min} < j+1 \leq I_{max}$	$j \neq \omega$ implies $I_{min} \leq j$, so clearly $I_{min} \leq j+1$. Because of test in node F, $j < n \leq I_{max}$, so $j+1 \leq n \leq I_{max}$.
	$1 \leq j+1 \leq n_0$	From top of loop, $1 \leq i_k$ and $j = 2 \times i_k$ so $2 \leq j$, thus $3 \leq j+1$. From test in node F, $j < n$, so $j+1 \leq n \leq n_0$ (see arc 7).
	$1 \leq j \leq n_0$	$2 \leq j$ and $j < n \leq n_0$.
	$M[j+1] \neq \omega$	True because we have chosen to require all elements of M to be defined on entry
	$M[j] \neq \omega$	(see arc 7).
11	$1 \leq j \leq n_0$	True from arc 8, where $1 \leq j+1 \leq n$.
	$M[j] \neq \omega$	$M[j+1] \neq \omega$ from arc 8.
15	$i \neq \omega$	Just assigned a value.
	$I_{min} \leq 2 \times i \leq I_{max}$	May not be true. We know that $i_{k+1} \leq n$ at this point, so push the assertion $2 \times n \leq I_{max}$ back to arc 1. Also see arc 3.

Proof of termination of loop.

Since i is monotonically increasing, the loop terminates if no semantic errors occur, i.e., if all the entry assertions are true.

We have now proven siftup to be free of semantic errors and infinite loops, if the following assertions are true on entry:

(a1)	$1 \leq i \leq n_0$	from arc 2
(a2)	$\forall\, 1 \leq \ell \leq n_0$, $M[\ell] \neq \omega$	from arcs 2 and 7
(a3)	$2 \leq I_{max}$	from arc 3
(a4)	$2 \times i \leq I_{max}$	from arc 3
(a5)	$1 \leq n \leq n_0$	from arc 7
(a6)	$2 \times n \leq I_{max}$	from arc 15

These are sufficient (although not quite necessary: some elements of M need not be defined) conditions for siftup to generate no semantic errors as it executes on the "real" machine we have assumed. Note that assertions a1 and a4 make assertion a3 redundant. Also, this set of assertions could be further simplified if the programmer stated the entry condition that is implied by the initial comment in siftup , $1 \leq i \leq n \leq n_0$. The above set of assertions represents hidden restrictions which are rarely included in the description of an algorithm. In particular, if treesort3 were used to sort an array of 30,000 elements on a machine with 16-bit 2's complement integers ($I_{max} = 32{,}767$) , and > 16-bit addresses (such as a large PDP-11 or a 360 with half-word integers), the statement

```
j := 2 x i;
```

could be executed with i = 30,000 , either generating an overflow or quietly assigning -5536 to j . Either the overflow would terminate

execution, or (worse), the comparison M[j+1] > M[j] would terminate on a subscript range error, or (worse yet, but somehow most likely) the assignment M[i] := M[j] would store into a random location in memory on the following iteration of the loop. Thus, a mathematically correct, certified program could generate complete garbage when run on a real machine.

Assertions for treesort3 (remember that n_0 is the value of n upon entry to treesort3).

Arc	Assertion	Proof
2	$n \neq \omega$	Value parameters are always defined.
3	$i \neq \omega$	Set in previous node.

Analysis of first loop:

$$n_{k+1} = n_k$$

$$i_{k+1} = i_k - 1$$

so n is invariant and i is monotonically decreasing.

4	$n \neq \omega$	From arc 2 and n invariant in loop.
4	$i \neq \omega$	True from previous node.
	$n \neq \omega$	From arc 2 and n invariant in loop.

Assertions for call to siftup , with arguments i and n (bound in treesort3) substituted for parameters i and n .

(1) $1 \leq i \leq n_0$	$2 < i$, but it is not immediately clear that $i \leq n_0$. Since i_k is monotonically decreasing, it is clear that if $i_1 \leq n_0$, all other i_k will be $\leq n_0$. But $i_1 = n \div 2$ and $2 \leq i_1$, so to enter the loop at all, $n_0 \geq 4$, in which case it is true that $n_0 \div 2 \leq n_0$.
(2) $\forall\ 1 \leq \ell \leq n_0, M[\ell] \neq \omega$	This must be an entry assumption for treesort3 . Move it to arc 1.
(3) $2 \times i \leq I_{max}$	Since $n = n_0$, this assertion is implied by (1) and (6).

Arc	Assertion	Proof
	(5) $1 \leq n \leq n_0$	$n = n_0$ and we noted above that to get to this arc at all, $n_0 \geq 4$.
	(6) $2 \times n \leq I_{max}$	This must be an entry assumption for **treesort3** . Move it to arc 1.
6	$I_{min} \leq i-1$	True because $i > 2$ and $I_{min} < 0$.

Analysis of second loop:

$$i_{k+1} = i_k - 1 \ .$$

n is invariant in loop.

Assertions for call to **siftup** , with arguments 1 and i substituted for parameters i and n .

10	(1) $1 \leq 1 \leq i$	True ; $i \geq 2$ at this point.
	(2) $\forall\ 1 \leq \ell \leq n_0 , M[\ell] \neq \omega$	Entry condition for **treesort3** , see arc 5.
	(5) $1 \leq i \leq n_0$	True; $i \geq 2$, $i_1 = n = n_0$, and i is monotonically decreasing.
	(6) $2 \times i \leq I_{max}$	True because $i \leq n_0$, and $2 \times n_0 \leq I_{max}$ is entry condition for **treesort3** (see arc 5).
11	[See analysis of exchange , below.]	
12	$I_{min} \leq i-1$	True. $2 \leq i$, $I_{min} < 0$.
also	$2 \leq I_{max}$	Move to arc 1, as an entry condition.

Termination of loops.

Since each loop has a monotonically decreasing variable, i , each terminates if it generates no semantic errors, i.e., if the entry conditions for **treesort3** are satisfied.

Assertions for exchange (as called from node J in **treesort3**).

Arc	Assertion	Proof
2	$1 \leq n_0$	True. $n_0 \geq 2$ to get to this call at all.
	$M[1] \neq \omega$	Entry condition for **treesort3** .
3	$1 \leq n_0$	True.
	$1 \leq i \leq n_0$	True, $2 \leq i \leq n_0$ within the **treesort3** loop.
	$M[i] \neq \omega$	Entry condition for **treesort3** .
4	$1 \leq i \leq n_0$	True.
	$t \neq \omega$	Defined in node C.

Conclusion

We have now proven **treesort3** to be free of semantic errors and infinite loops if executed on the machine described and if the following assertions are true on entry:

(1) $\forall\ 1 \leq \ell \leq n_0, M[\ell] \neq \omega$ from arc 5

(2) $2 \times n_0 \leq I_{max}$ from arc 5

(3) $2 \leq I_{max}$ largest constant

The second assertion is a hidden restriction which will prevent the proper execution of the algorithm on large arrays on a machine with larger addresses than integers (such as a large PDP-11 or a 360 with half-word integers). Note in passing that **siftup** has the entry condition that $1 \leq n_0$, but treesort3 does not require $1 \leq n_0$. A quick examination

of **treesort3** shows that it works quite properly in this degenerate case, skipping both loops and returning.

The only difficult parts in the morass of detailed proofs were: (a) the proof that i is monotonically increasing in the main loop of **siftup** , in which we used the overly-stringent assumption that $i > 1$; **and** (b) the proof at arc 8 of **siftup** that $1 \leq j+1 \leq n_0$, which used global information about the behavior of n inside the loop.

APPENDIX: Clean termination of **treesort3** on CDC 6600.

Because **comparisons** may generate overflow, the following additional assertions are required:

Node	Assertion	Proof
		(These proofs assume the additional constraint that I_{min} , I_{max} , Rmin , and R_{max} are symmetrical about zero, i.e., that $I_{min} = -I_{max}$, $R_{min} = -R_{max}$.)
siftup		
E	$I_{min} \leq n-j \leq I_{max}$	j and n are both ≥ 1 (from node C), and the difference of two numbers with the same sign cannot overflow.
F	$I_{min} \leq n-j \leq I_{max}$	Same.
G	$R_{min} \leq M[j+1]-M[j] \leq R_{max}$	Cannot be proved without a restriction like $\forall\, 1 \leq \ell \leq n_0 \;\; R_{min} \div 2 \leq M[\ell] \leq R_{max} \div 2$ or $\forall\, 1 \leq \ell \leq n_0 \;\; 0 \leq M[\ell] \leq R_{max}$ This is an additional entry restriction on **treesort3** . This subtraction may also generate an underflow if M[j+1] and M[j] are very small and almost equal.
I	$R_{min} \leq M[j] \leq R_{max}$	See node G above.

Node	Assertion	Proof
treesort3		
D	$I_{min} \leq i-2 \leq I_{max}$	On arc 3: $I_{min} \div 2 < i < I_{max} \div 2$ from node C, so the assertion is true if $I_{min} \leq (I_{min} \div 2)-2$, i.e., if $I_{min} \leq -3$. Although not very interesting, this is a valid entry restriction on treesort3 . A more interesting one would be $n > 2$. On arc 7: since $i > 1$ at this point, the assertion is true because of our initial assumption that $I_{min} < 0$.
H	$I_{min} \leq i-2 \leq I_{max}$	On arc 8: since $i = n$, this becomes the entry condition $I_{min} < n-2$. On arc 13: since $i > 1$ at this point, the assertion is always true.

Conclusion

treesort3 will terminate cleanly on a CDC 6600 if the following additional restrictions are true on entry:

(1) $\forall$ i,j such that $1 \leq i < n_0$ and $1 \leq j < n_0$,

$$R_{min} \leq M[i] - M[j] \leq R_{max} .$$

Two sufficient forms of this are:

$$\forall\ 1 \leq \ell \leq n_0 , \quad R_{min} \div 2 \leq M[\ell] \leq R_{max} \div 2$$

and

$$\forall\ 1 \leq \ell \leq n_0 , \quad 0 \leq M[\ell] \leq R_{max} .$$

(2) $I_{min} \leq n-2$ and $I_{min} \leq -3$.

Acknowledgment

Don Knuth suggested the shortcut for proving loop termination.

References

[1] Floyd, R. W., "Algorithm 245, TREESORT3," Comm. ACM 7 (Dec. 1964), 701.

[2] London, R. L., "Certification of Algorithm 245," Comm. ACM 13 (June 1970), 371.

[3] Redish, K. A., "Comment on London's Certification of Algorithm 245," Comm. ACM, January 1971, pp. 50-51.

```
  400 CONTINUE
      TEMP = C*E(NUM1) + S*D(NU)
      D(NU) = -S*E(NUM1) + C*D(NU)
      E(NUM1) = TEMP
      GO TO 210
C
C     INTERNAL PROCEDURE TO CALCULATE THE ROTATION CORRESPONDING TO
C     THE VECTOR(P,Q).
C
  500 PP = ABS(P)
      QQ = ABS(Q)
      IF(QQ .GT. PP) GO TO 510
      NORM = PP*SQRT(1. + (QQ/PP)**2)
      GO TO 520
  510 IF(QQ .EQ. 0.) GO TO 530
      NORM = QQ*SQRT(1. + (PP/QQ)**2)
  520 C = P/NORM
      S = Q/NORM
      GO TO RETURN,(310,340,360)
  530 C = 1.
      S = 0.
      NORM = 0.
      GO TO RETURN,(310,340,360)
      END
```

CERTIFICATION OF ALGORITHM *245* [M1] TREESORT 3 [Robert W. Floyd, *Comm. ACM* **7** (Dec. 1961), 701]: PROOF OF ALGORITHMS-A NEW KIND OF CERTIFICATION

RALPH L. LONDON * (Recd. 27 Feb. 1969 and 8 Jan. 1970)
Computer Sciences Department and Mathematics Research Center, University of Wisconsin, Madison, WI 53706

* This work was supported by NSF Grant GP-7069 and the Mathematics Research Center, US Army under Contract Number DA-31-124-ARO-D-462.

ABSTRACT: The certification of an algorithm can take the form of a proof that the algorithm is correct. As an illustrative but practical example, Algorithm 245, ***TREESORT 3*** for sorting an array, is proved correct.

KEYWORDS AND PHRASES: proof of algorithms, debugging, certification, metatheory, sorting, in-place sorting
CR CATEGORIES: 4.42, 4.49, 5.24, 5.31

Certification of algorithms by proof. Since suitable techniques now exist for proving the correctness of many algorithms [for example, 3-7], it is possible and appropriate to certify algorithms with a proof of correctness. This certification would be in addition to, or in many cases instead of, the usual certification. Certification by testing still is useful because it is easier and because it also provides, for example, timing data. Nevertheless the existence of a proof should be welcome additional certification of an algorithm. The proof shows that an algorithm is debugged by showing conclusively that no bugs exist.

It does not matter whether all users of an algorithm will wish to, or be able to, verify a sometimes lengthy proof. One is not required to accept a proof before using the algorithm any more than one is expected to rerun the certification tests. In both cases one could depend, in part at least, upon the author and the referee.

As an example of a certification by proof, the algorithm *TREESORT* 3 [2] is proved to perform properly its claimed task of sorting an array $M[1:n]$ into ascending order. This algorithm has been previously certified [1], but in that certification, for example, no arrays of odd length were tested. Since *TREESORT 3* is a fast practical algorithm for in-place sorting and one with sufficient complexity so that its correctness is not immediately apparent, its use as the example is more than an abstract exercise. It is an example of considerable practical importance.

Outline of TREESORT 3 and method of proof. The algorithm is most easily followed if the array is viewed as a binary tree. $M[k \div 2]$ is the parent of $M[k]$, $2 \leq k \leq n$. In other words the children of $M[j]$ are $M[2j]$ and $M[2j+1]$ provided one or both of the children exist.

The first part of the algorithm permutes the M array so that for a segment of the array, each parent is larger than both of the children (one child if the second does not exist). Each call of the auxiliary procedure *siftup* enlarges the segment by causing one more parent to dominate its children. The second part of the algorithm uses *siftup* to make the parents larger over the whole array, exchanges $M[1]$ with the last element and repeats on an array one element shorter. The above statements are motivation and not part of the formal proof.

That ***TREESORT 3*** is correct is proved in three parts. First the procedure *siftup* is shown to perform as it is formally defined below. Then the body of ***TREESORT*** 3, which uses *siftup* in two ways, is shown to sort the array into ascending order. (The proof of the procedure *exchange* is omitted.) The proofs are by a method described in [3, 4, 7]: assertions concerning the progress of the computation are made between lines of code, and the proof consists of demonstrating that each assertion is true each time control reaches that assertion, under the assumption that the previously encountered assertions are true. Finally termination of the algorithm is shown separately.

The lines of the original algorithm have been numbered and the assertions, in the form of program comments, are numbered correspondingly. The numbers are used only to refer to code and to assertions and have no other significance. One extra begin-end pair has been inserted into the body of *TREESORT 3* in order that the control points of two assertions (3.1 and 4.1) could be distinguished. In *siftup* the assertions **10.1 and 10.2 express** the correct result; in the body of ***TREESORT 3*** the assertions 9.3 and 9.4 do likewise.

Definition of siftup and notation. We now define formally the procedure *siftup*(i,n), where n is a formal parameter and not the length of the array M. Let $A(s)$ denote the set of inequalities $M[k \div 2] \geq M[k]$ for $2s \leq k \leq n$. (If $s > n \div 2$, then $A(s)$ is a vacuous statement.) If $A(i+1)$ holds before the call of *siftup*(i,n) and if $1 \leq i \leq n \leq$ *array size*, then after *siftup*(i,n):

(1) A (i) holds;

(2) the segment of the array $M[i]$ through $M[n]$ is permuted; and

(3) the segment outside $M[i]$ through $M[n]$ is unaltered.

In order to prove these properties of *siftup*, some notation is required. The formal parameter i will be changed inside *siftup*. Since i is called by value, that change will be invisible outside *siftup*. Nevertheless it is necessary to use the initial value of i as well as the current value of i in the proof of *siftup*. Let i_0 denote the value of i upon entry to *siftup*.

Similarly let M_0 denote the M array upon entry to *siftup*. The notation "$M = p(M_0)$ with M := *copy*" means "If $M[i]$:= *copy* were done, M is some permutation of M_0 as described in (2) and (3) of the definition of *siftup*." "$M = p(M_0)$" means the same without the reference to $M[i]$:= *copy* being done.

Code and assertions for siftup.

```
0   procedure siftup(i, n); value i, n; integer i, n ;
1   begin real copy; integer j;
      comment
        1.1:  1  ≤ i0 = i ≤ n ≤ array size
        1.2: A (i0+1)
        1.3:  M = p(M0);
```

```
2     copy := M[i];
3     loop: j := 2 X i;
      comment
        3.1: i ≤ n
        3.2: 2i = j
        3.3: i = i0 or i ≥ 2i0
        3.4: M = p(M0) with M[i] := copy
        3.5: A(i0) or (i = i0 and A(i0+1))
        3.6: M[i÷2] > copy or i = i0
        3.7: M[i÷2] ≥ M[i] or i = i0;
4     if j ≤ n then
5     begin if j < n then
6a      begin if M[j+1] > M[j] then
6b        j := j + 1 end;
        comment
          6.1: i = j ÷ 2
          6.2: 2i ≤ j ≤ n
          6.3: i = i0 or i ≥ 2i0
          6.4: M = p(M0) with M[i] := copy
          6.5: A(i0) or (i = i0 and A(i0+1))
          6.6: M[i÷2] > copy or i = i0
          6.7: M[i÷2] ≥ M[i] or i = i0
          6.8: (2i < n and M[j] = max(M[2i], M[2i+1])) or
               (2i = n and M[j] = M[n])
          6.9: M[i] ≥ M[j] or i = i0;
7       if M[j] > copy then
8a      begin M[i] := M[j];
          comment
            8.1: i = i0 or i ≥ 2i0
            8.2: 2i ≤ j ≤ n
            8.3: M[j÷2] = M[i] = M[j] > copy
            8.4: M[i÷2] ≥ M[j] or i = i0
            8.5: M = p(M0) with M[j] := copy
            8.6: A(i0);
8b        i := j;
          comment
            8.7: i ≥ 2i0
            8.8: i = j ≤ n
            8.9: M[i÷2] > copy
            8.10: M[i÷2] ≥ M[i]
            8.11: M = p(M0) with M[i] := copy
            8.12: A(i0);
8c      go to loop end
9     end;
      comment
        9.1: M[j] ≤ copy if reached from 7 or
          2i = j > n if reached from 4;
10    M[i] := copy;
      comment
        10.1: Al = p(M0)
        10.2: A(i0);
11    end siftup;
```

Verification of the assertions of siftup. Reasons for the truth of each assertion follow:

1.1-1.2: Assumptions for using *siftup*.

1.3: p is the identity permutation.

3.1-3.7: If reached from 2,
3.1: 1.1.
3.2: 3.
3.3, 3.5-3.7: i = i_0 by 1.1. 3.5 also requires 1.2.
3.4: 1.3 and 2.
If reached from 8, respectively, 8.8, 3, 8.7, 8.11, 8.12, 8.9 and 8.10.

6.1: At3.2 j = 2i and by 6b, j might be 2i + 1. $i = j \div 2$ in either case.

6.2: After 4, $j \leq n$. j is altered from 3.1 to 6.2 only at 6b. Before 6b, j < n by 5. Hence $j \leq n$ at 6.2. $2i \leq$ j by 6.1.

6.3-6.7: 3.3-3.7, respectively.

6.8: If 4 is true and 5 is false, j = 2i = n (using 3.2) so the second clause of 6.8 holds. If 4 is true and 5 is true, then at 6a, 2i = j < n (using 3.2) so $M[j+1] = M[2i+1]$ is defined. Now at 6.8, j = $2i$ or j = $2i+1$. In either case, by 6a and 6b, the first clause of 6.8 holds.

6.9: By 6.5 $i \neq i_0$ gives $A(i_0)$. $2i_0 \leq 2i \leq j \leq n$ by 6.3 and 6.2. Hence $A(i_0)$ and 6.1 give $M[i] = M[j \div 2] \geq M[j]$.

8.1: 6.3.

8.2: 6.2.

8.3: i = $j \div 2$ by 6.1, $M[i] = M[j]$ by 8a and $M[j] >$ *copy* by 7.

8.4: 6.7 nnd 6.9.

8.5: 6.4 requires that $M[i]$ be replaced by *copy*. Since $M[i] = M[j]$ by 8a, $M[j]$ may equally well be replaced with *copy*. 8.1 and 8.2 give $i_0 \leq i \leq n$ so that the change to M at 8a is in the segment $M[i_0]$ through $M[n]$.

8.6: By 83 and if 6.8 (first clause) holds, $M[i] \geq M[2i]$ and $M[i] \geq M[2i+1]$. By 8a and if 6.8 (second clause) holds, $M[i] = M[j] = M[n] = M[2i]$ and $M[2i+1]$ does uot exist for this call of *siftup*. $A(i_0+1)$ holds at 6.5 since $A(i_0)$ implies $A(i_0+1)$. If $i = i_0$, $A(i_0+1)$ and the relations above on $M[i]$ give $A(i_0)$. If $i \neq i_0$, then 8a, 8.4, $A(i_0)$ at 6.5 and the relations above on $M[i]$ give $A(i_0)$ at 8.6.

8.7: 8b, 8.1 and 8.2.

8.8: 8b and 8.2.

8.9: 8b and 8.3.

8.10: At 8.6, $2i_0 \leq j \leq n$ by 8.1 and 8.2. Hence by 8.6, $M[j \div 2] \geq M[j]$. Use 8b on $M[j \div 2] \geq M[j]$.

8.11: 8b and 8.5.

8.12: 8.6.

9.1: 9.1 is reached only if 7 is false or if 4 is false. 2i = j by 3.2.

10.1-10.2: If reached from 7,
10.1: 6.4 and 10. (6.2 and G.3 give $i_0 \leq i \leq n$ ensuring the change to M at 10 is in the segment $M[i_0]$ through $M[n]$.).
10.2: By 10, 9.1, G.2 and 6.8, $M[i] = copy \geq M[j] \geq M[2i]$ and, if $M[2i+1]$ esists, $M[j] \geq M[2i+1]$. If $i = i_0$, 10.2 follows as in 8.6. Ii $i \neq i_0$, 6.6 and 10 give $M[i \div 2] > copy = M[i]$. A (i_0) at 6.5 now gives $A(i_0)$ at 10.2.
If reached from 4,
10.1: 3.4 and 10. (3.1 and 3.3 give $i_0 \leq i \leq n$.)
10.2: 2i > n means no relations in $A(i_0)$ of the form $M[i] \geq \cdots$. If i = i_0, 3.5 gives 10.2. If $i \neq i_0$, 3.6 and 10 give $M[i \div 2] >$ copy $= M[i]$. $A(i_0)$ at 3.5 now gives 10.2.

Code and assertions ***for the body of TREESORT 3.***

```
0  integer i;
     comment
       0.1: A(n÷2+1);
1    for i := n÷2 step -1 until 2 do
2    begin
       comment
         2.1: A(i+1)
         2.2: Assumptions of siftup satisfied;
3      siftup(i,n);
       oommcnt
         3.1: A(i);
4    end;
     comment
       4.1: M[p] ≤ M[p+1] for n + 1 ≤ p ≤ n − 1
       4.2: A(2), i.e. M[k÷2] ≥ M[k] for 4 ≤ k ≤ n;
5    for i := n etep -1 until 2 do
6    begin
       comment
         6.1: M[p] ≤ M[p+1] for i + 1 ≤ p ≤ n − 1
         6.2: M[k÷2] ≥ M[k] for 4 ≤ k ≤ i
         6.3: M[i+1] ≥ M[r] for 1 ≤ r ≤ i
         6.4: Assumptions of siftup satisfied;
```

```
7    siftup (1, i);
     comment
       7.1: M[p] ≤ M[p+1] for i + 1 ≤ p ≤ n − 1
       7.2: M[k÷2] ≥ M[k] for 2 ≤ k ≤ i
       7.3: M[1] ≥ M[r] for 2 ≤ r ≤ i
       7.4: M[i+1] ≥ M[1];
8    exchange (M[1], M[i]);
     comment
       8.1: M[i] ≥ M[r] for 1 ≤ r ≤ i − 1
       8.2: M[p] ≤ M[p+1] for i ≤ p ≤ n − 1
       S.3: M[k÷2] ≥ M[k] for 4 ≤ k ≤ i − 1;
9  end;
   comment
     0.1: M[p] ≤ M[p+1] for 2 ≤ p ≤ n − 1
     9.2: M[2] ≥ M[1]
     9.3: M[p] ≤ M[p+1] for 1 ≤ p ≤ n − 1, i.e. M is fully
       ordered
     9.4: M is a permutation of M0;
```

Verification of the assertions for the body of TREESORT 3. Reasons for the truth of each assertion follow:

0.1: Vacuous statement since $2(n \div 2+1) > n$.

2.1: If reached from 0.1, by 1 substitute $i = n \div 2$ in 0.1.
If reached from 3.1, by 1 substitute $i = i + 1$ in 3.1 to account for the change in i from 3.1 to 2.1.

2.2: 2.1, the bound on i implied by 1 and the array size being n.

3.1: 2.1 and the definition of *siftup*(i, n).

4.1: Vacuous statement.

4.2: If $n \geq 4$, 3 is executed; hence 3.1 with $i = 2$. If $n \leq 3$, vacuous statement.

6.1-6.3: If reached from 4.1,
G.1-6.2: By 5 substitute $i = n$ in 4.1 and 4.2.
6.3: Vacuous statement for i = n.
If reached from 8.1, by 5 substitute $i = i + 1$ in 8.2, 8.3 and S.1, respectively.

6.4: 5 and 6.2, i.e. A (2) for the subarray $M[1:i]$.

7.1: G.1 and (3) of *siftup*.

7.2: 6.2 and (1) of *siftup*.

7.3: 7.2 noting that $M[1] = M[k \div 2]$ if $k = 2$ and using the transitivity of $\geq$.

7.4: Vacuous for i = n. Otherwise 6.3 for the appropriate r since by (2) of *siftup*, $M[1]$ at 7.3 is one of the $M[r]$, $1 \leq r \leq i$, at 6.3.

8.1: 7.3 with the changes caused by 8 (only $M[1]$ and $M[i]$ are altered by 8).

8.2: By 8 substitute $M[i]$ for $M[1]$ in 7.4; then 7.1 also holds for p = i.

8.3: 7.2 excluding only the one or two relations $M[1] \geq \ldots$, and the one relation $\ldots \geq M[i]$.

9.1-9.3: If $n \geq 2$, 8 is executed;
9.1: 8.2 with $i = 2$.
9.2: 8.1 with $i = 2$.
9.3: 9.1 and 9.2.
If $n \leq 1$, 9.1-9.3 are vacuous statements.

9.4: The only operations done to M are *siftup* and *exchange* all of which leave M as a permutation of M_0.

Proof of termination of TREESORT 3. Provided *siftup* and *exchange* terminate, it is clear that *TREESORT 3* terminates. Note that each parameter of *siftup* is called by value so that i is not changed in the body of the for loops.

The procedure *exchange* certainly terminates. In *siftup* the only possibility for an unending loop is from 3 to 8b and back to 3. Note that all changes to i (only at 8b) and to j (only at 3 and Gb) occur in this loop and that on each cycle of this loop both i and j are changed. By the test at 4, it is sufficient to show that j strictly increases in value. $i \geq 1$ means $2i > i$. At 8b, $j = i < 2i$ while at 3, $j = 2i$, i.e. j(at 3) = 2i > i = j(at 8b). Hence each setting to j at 3 strictly increases the valuo of j. The only other setting to j (at 6b), if made, eimilarly increases the value of j.

REFERENCES:

1. ABRAMS, I'. S. Certification of Algorithm 245. *Comm. ACM 8* (July 1965), 445.
2. FLOYD, 11. W. Algorithm 245, TREESORT 3. *Comm. ACM* 7 (Dec. 1964), 701.
3. FLOYD, R. W. Assigning meanings to programs. Proc. of a Symposium in Applied Mathematics, Vol. 19 Mathematical Aspects of Computer Science, J. T. Schwartz (Ed.), American Mnth. Society, Providence, 11. I., 1967, pp. 19-X.
4. KNUTH, D. E. *The Art of Computer Programming, Vol. 1—Fundamental Algorithms.* Addison-Wesley, Reading, Mass., 1968, Sec. 1.2.1.
5. MCCARTHY, J. A basis for a mathematical theory of computation. In *Computer Programming and Formal Systems*, P. Braffort and D. Hirschberg (Eds.), North Holland, Amsterdam, 1963, pp. 33-70.
6. MCCARTHY, J., AND PAINTER, J. A. Correctness of a compiler for arithmetic expressions. Proc. of a Symposium in Applied Mathematics, Vol. 19—Mathematical Aspects of Computer Science, J. T. Schwartz (Ed.), American Math. Society, Providence, R. I., 1967, pp. 33-41.
7. NAUR, P. Proof of algorithms by general snapshots. *BIT 6* (1966), 310-316.

REMARK ON ALGORITHM *201* [M1] SHELLSORT [J. Boothroyd, *Comm. ACM 6* (Aug. 1963), 445]

J. P. CHANDLER AND W. C. HARRISON* (Recd. 19 Sept. 1969)

Department of Physics, Florida State University, Tallahassee, FL 32306

* This work was supported in part by AEC Contract No. AT-(40-1)-3509. Computational costs were supported in part by National Science Foundation Grant GJ 367 to the Florida St ate University Computing Center.

KEY WORDS AND PHRASES: sorting, minimal storage sorting, digital computer sorting

CR CATEGORIES: 5.31

Hibbnrd [1] hns coded this method in a way that increases the speed significantly. In SHELLSORT, each st age of each sift consists of successive pair swaps. The modificat ion replaces each set of n pair swaps by one "save," n − 1 moves, and one in. ert ion.

Table I gives timing information for ALGOL, FORTR AN, and COMPASS (assembly language) versions of SHELLSORT and the

TABLE I. SORTING TIMES IN SECONDS FOR 10,000 RANDOMLY ORDERED NUMBERS ON THE CDC 6400 COMPUTER

Algorithm	*Source Language*		
	ALGOL	FORTRAN	COMPASS
SHELLSORT	53.40	7.1s	2.3s
SHELLSORT	36.56	5.98	1.87

PROVING CLEAN TERMINATION OF COMPUTER PROGRAMS

Richard L. Sites
Computer Science Department
Stanford University
Stanford, California 94305

Abstract

This paper presents a system for proving that a computer program contains no semantic errors and no infinite loops, and hence that it always terminates cleanly. This work differs from other work on verification of program correctness in two important ways: (1) it deals explicitly with the finite limitations of real machines, and (2) it does not examine what the program accomplishes; no description of the correctness properties of the program is required. A recent ALGOL program for computing medians is used as a running example.

Keywords and Phrases: proof of termination, proof of correctness

CR Categories: 5.24

Much of the theoretical work in verifying program correctness has concentrated on theorem proving techniques, formal language schemata, formal logic, program synthesis, and program equivalence. A common theme in this work is the process of describing a program by a set of assertions, and then inductively proving that the assertions are true. In such an approach, the assertions (or at least the important ones) are usually supplied by a human, then the verification system tries to prove them. At the successful conclusion of this process, the program has been proved to do exactly what the assertions describe [Floyd], [King], [Good].

One of the drawbacks of this approach is that it takes a lot of effort to create the proper assertions -- to find assertions which describe both what the program actually does and what it is intended to do. It is easy to write down assertions which loosely describe what the program does, but which happen to fail in degenerate cases (such as the first time through a loop, or a normally positive variable starting out exactly zero); it is also easy to write down assertions which do not fully describe the intended functioning of the program, so that the program may be carefully proved to work as the assertions describe, but it still would contain "bugs" in actual use. For example, the correctness of a program to sort elements 1 through n of an array A might be described with a final assertion like this:

$$\forall\ 1 \leq \ell \leq n-1 \quad , \quad A[\ell] \leq A[\ell+1] \quad .$$

While this is a perfectly reasonable description of the intended function of the sort program, the following program can also be rigorously shown to work as the assertion describes:

```
for i := 1 until n do
   A[i] := 37;
```

Another problem with programs that have been proved correct is that the proof applies only to ideal machines whose numbers have unlimited precision and range. When run on real, finite, computers, such programs may deliver improper results even after they have been rigorously certified (see for example [Sites]).

Large "real-world" programs (such as a compiler) are usually developed to the point that they appear to go through all the right motions, that they basically work, and then the program enters a shakedown period during which many test cases are run and perhaps new users are allowed to try the program. The purpose of this shakedown is to eliminate most anomalies and to improve the confidence level that the program is working properly.

The rest of this paper describes a system to make this shakedown process more rigorous and to detect errors due to the finite limitations of real machines. Programs exhibit bugs in one of two ways: they produce incorrect results, or they terminate abnormally. Correct results are sometimes hard to describe rigorously (although there is a high payoff in describing simple consistency checks), but abnormal termination can be more precisely specified. In fact, we have no good notation for describing what it means for a complicated program to be correct; many data processing concepts, such as "this compiler produces correct object code", have no simple rigorous form. A system which tries to prove that a program will always terminate normally could be quite useful for increasing confidence in the proper functioning of a large program. The system would say nothing about what the program does (i.e., sort an array);

instead, such a system would report that whatever the program does, it terminates cleanly. The system would verify that the program contains no semantic errors or infinite loops -- no overflows, out-of-range subscripts, references to null or undefined pointers, etc.

While proving that a program terminates is in general an unsolvable problem, most real programs are intended to terminate and have good reasons for doing so. Therefore, it is reasonable to expect that automatic means could be used to prove termination of many useful programs.

To make this concept more specific, let us consider a version of a nontrivial program written by R. L. Rivest [Rivest and Floyd]. This example will be used throughout the paper to illustrate the techniques presented. See Program on next page.

To prove that this program terminates cleanly, it is necessary to prove, for example, that line 19 produces no overflow; that in line 26, i is defined and in the proper subscript range for X ; and that the loop at lines 28-29 terminates (without a bad subscript).

This paper discusses many of the issues in creating a mechanical proof that Rivest's program terminates cleanly. It does not, however, go into much detail about theorem proving techniques, or about the specific theorems of Rivest's program. It will not deal with the recursive call to select at line 17; essentially, the discussion below will only treat the functioning of select when $r-\ell \leq 600$. The dotted paths on the flow graph below are intended to indicate that as an inexpensive byproduct of the techniques presented here, some cases of recursion can be shown to terminate by treating the recursive call as an assignment to the parameters and a branch to the beginning of the program. Of course, treating the call as a branch does not deal with what happens when the call returns.

```
1.   procedure select(X,ℓ,r,k); value ℓ,r,k; array X;
2.   comment  select will rearrange the values of X[ℓ:r] so that
3.      X[k] contains the (k-ℓ+1)st smallest value.
4.      ℓ ≤ i ≤ k implies X[i] ≤ X[k] and
5.      k ≤ i ≤ r implies X[i] ≥ X[k];
6.   begin integer n, i, j, f, s, sd, ℓℓ, rr, t;
7.   while r-ℓ > 0 do
8.      begin
9.      if r-ℓ > 600 then
10.        begin
11.        n := r-0+1;
12.        i := k-ℓ;
13.        s := entier( 0.5 * exp(2*ln(n)/3) );
14.        sd := entier( 0.5 * sqrt(ln(n)*s*(n-s)/n) * sign(i-n/2) );
15.        ℓℓ := max( ℓ, k - i*s/n + sd );
16.        rr := min( r, k + (n-i)*s/n + sd );
17.        select(X,ℓℓ,rr,k)
18.        end;
19.     i := ℓ + 1;
20.     j := r - 1;
21.     t := X[k];
22.     X[k] := X[ℓ];
23.     X[ℓ] := t;
24.     if X[r] < t then
25.        exchange( X[r], X[ℓ] );
26.  P: while X[i] < t do
27.        i := i + 1;
28.     while X[j] > t do
29.        j := j - 1;
30.     if i < j then
31.        begin
32.        exchange( X[i], X[j] );
33.        i := i + 1;
34.        j := j - 1;
35.        go to P
36.        end;
37.     if X[ℓ] = t then
38.        exchange( X[ℓ], X[j] )
39.     else begin
40.        j := j + 1;
41.        exchange( X[j], X[r] )
42.        end;
43.     if j ≤ k then
44.        ℓ := j + 1;
45.     if k ≤ j then
46.        r := j - 1
47.     end
48.  end select
```

Program

This paper will not deal with lines 11-16 because they involve floating-point numbers, which the prototype system is not prepared to handle, and for which exposure to overflow, underflow, and division by zero are much harder to avoid than for integers.

In summary, proofs of clean termination are useful for several reasons:

(1) This technique may be economically applicable to a larger set of programs than more exacting proofs of correctness.

(2) Machines are better than humans at mechanically examining degenerate cases; it is difficult to create correctness assertions which are true in all cases including the degenerate ones.

(3) By examining programs run on real machines, with finite precision and finite-range arithmetic, the system deals with a source of bugs which other correctness techniques don't address at all.

(4) If the program cannot be proved to terminate cleanly, then the proof process should detail which parts of the program will always terminate and which may not, thus focusing the user's attention on the complicated, error-prone, or interesting part of a program.

(5) For programs like operating system subsystems, it is often desirable that the program always return control to the operating system, even if it sometimes gives incorrect results.

I. Flow Graph Processing

Let us now take Rivest's program and apply to it a mechanical process which is often able to prove clean termination. We start by viewing the program as a flow graph (Figure 1).

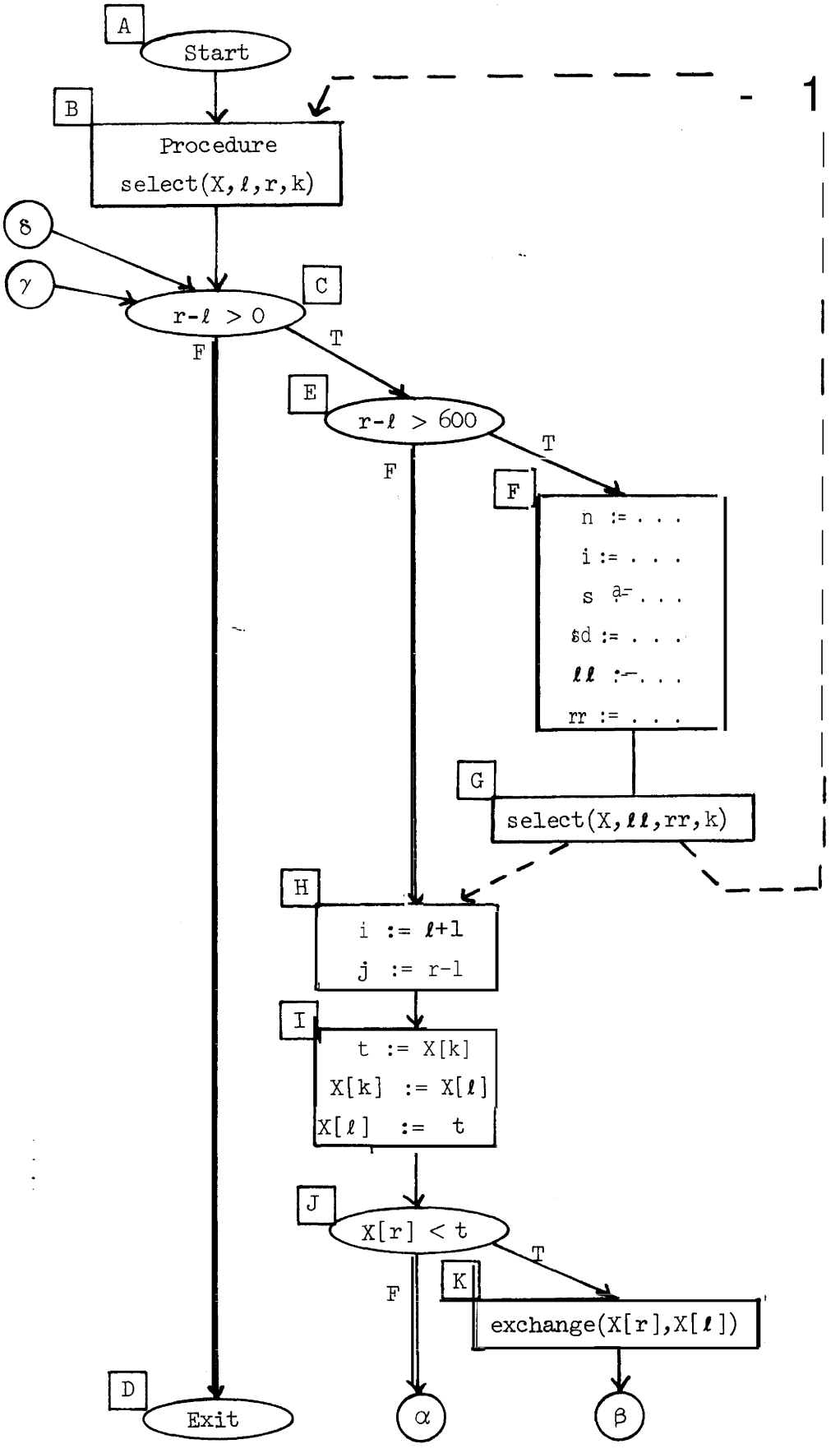


Figure 1. The flow graph for Rivest's program

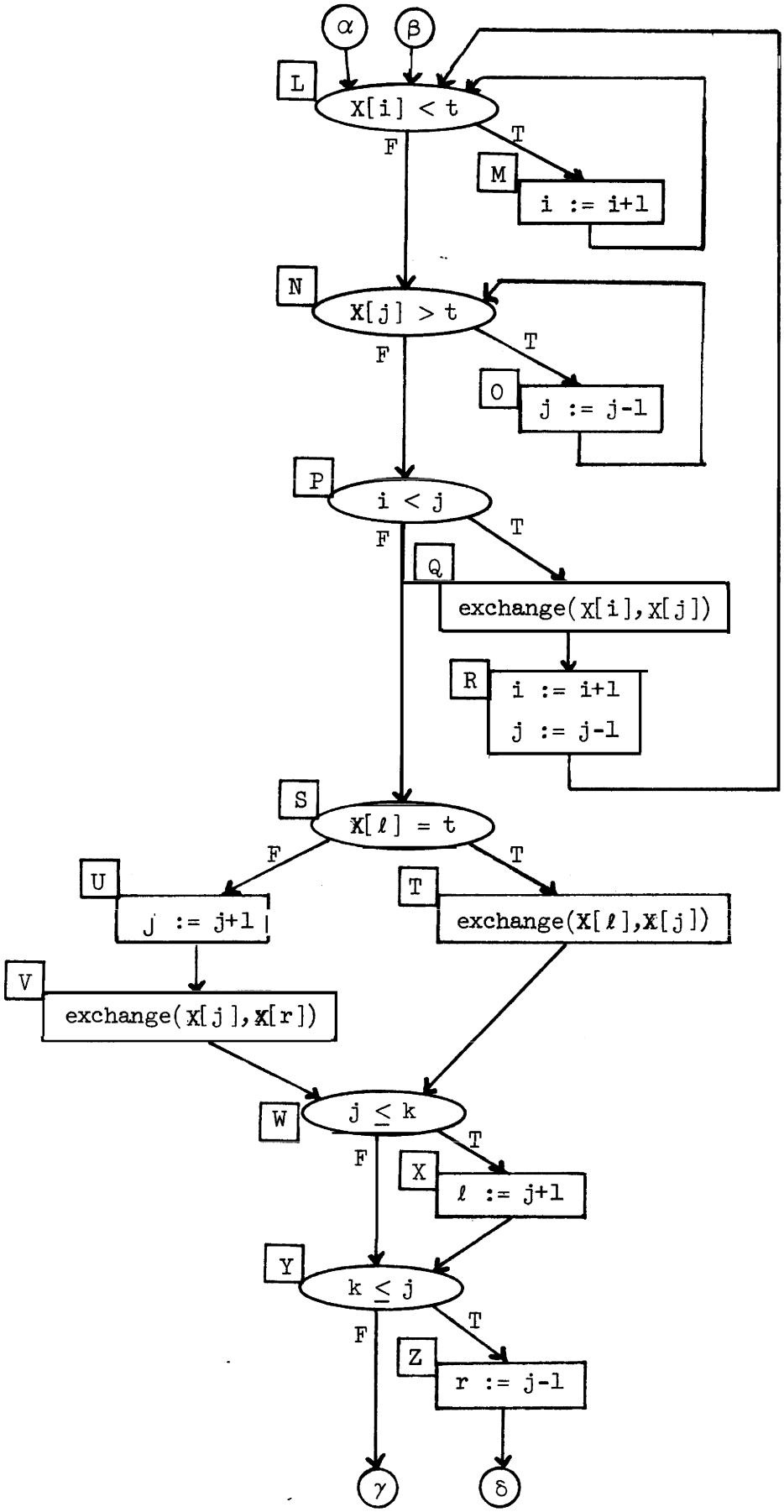


Figure 1 continued.

In order to make analysis of the loops more manageable, the flow graph is modified according to the following set of rules.

First, we perform interval analysis [Allen a,b], [Allen and Cocke], [Cocke], [Cocke and Schwartz] on the flow graph, and do any necessary node splitting so that every loop has a single entry node. The point of forcing all loops to have a single entry node is to avoid situations like the one in Figure 2a.

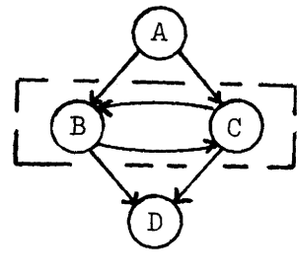


Figure 2a. Flow graph which needs node splitting.

where it is impossible to determine the state of the program when entering node B without having first examined the state of the program when leaving node B . Node splitting produces the modified graph in Figure 2b,

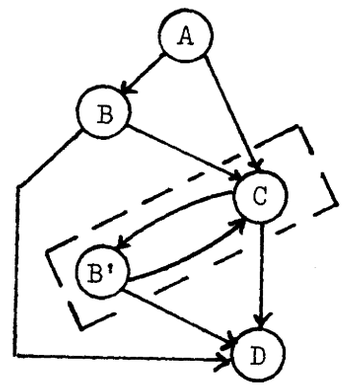


Figure 2b. Same graph after node splitting.

in which the loop now has a single entry node, C .

When applied to Figure 1, interval analysis leads to successive reductions as shown in Figure 3. No node splitting is required.

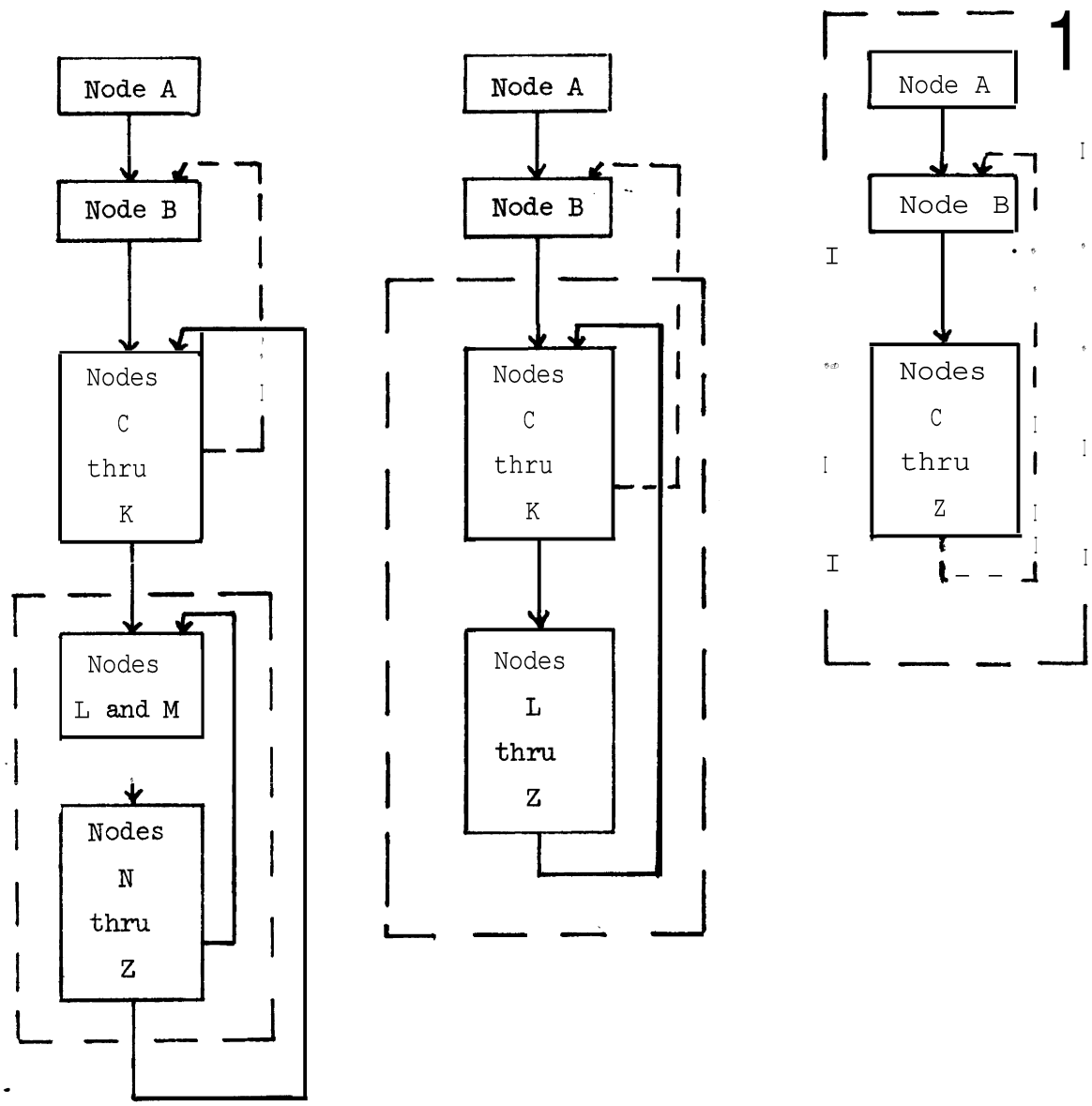


Figure 3. Interval analysis of Rivest's program.

Second, at the head of each interval which is a loop, we add a "loophead" node and reroute all the latchback arcs (arcs which branch back to the head of the interval) and initial entry arcs through this new node. The loophead node gives us a convenient place to attach loop induction information and loop termination assertions. This step generates four loophead nodes in our example, see Figure 4. (For simplicity, we shall henceforth ignore the dotted arcs, which correspond to the recursion.)

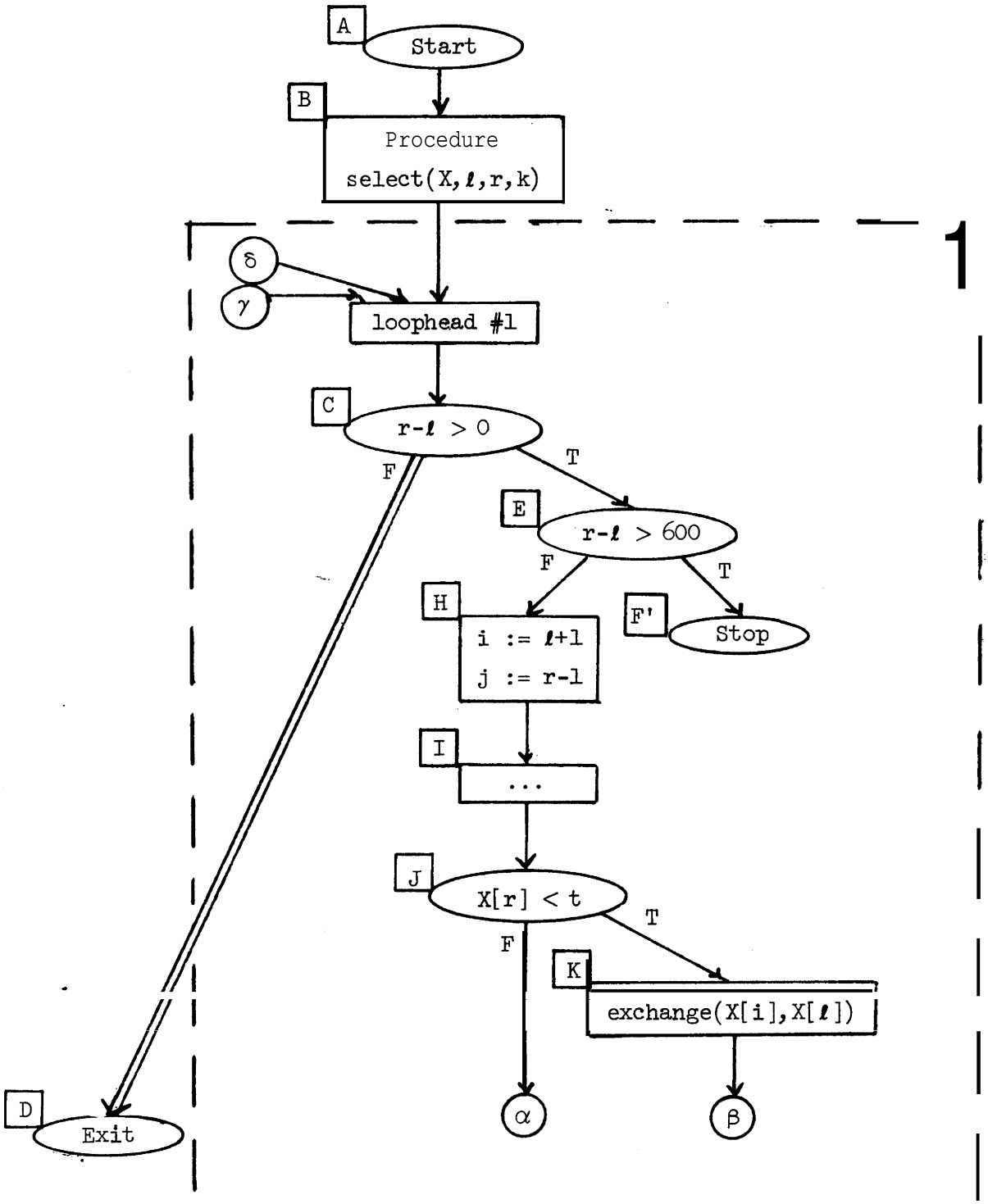


Figure 4. Flow graph after interval analysis and insertion of "loophead" nodes. Double lines are loop exit arcs.

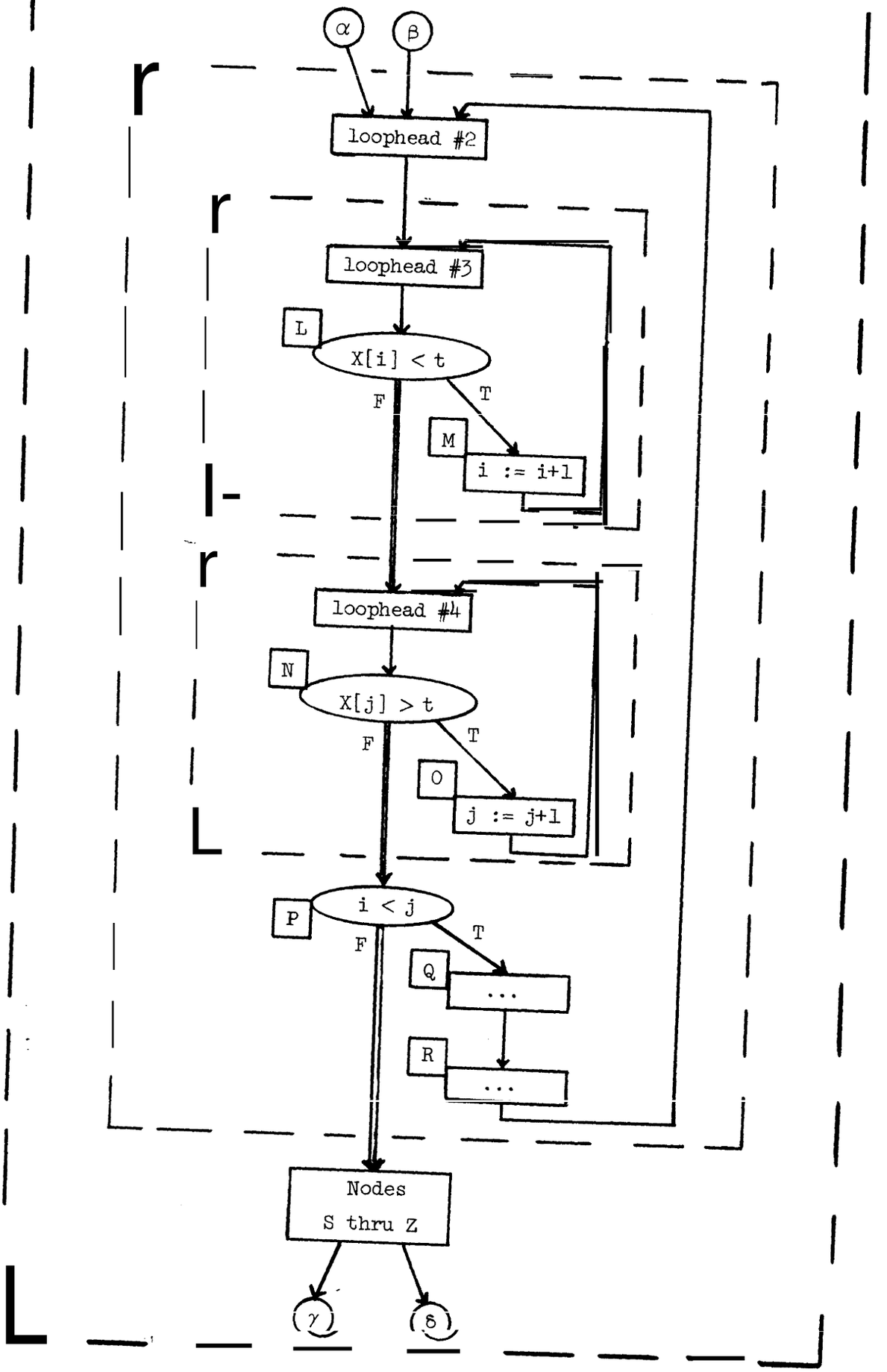
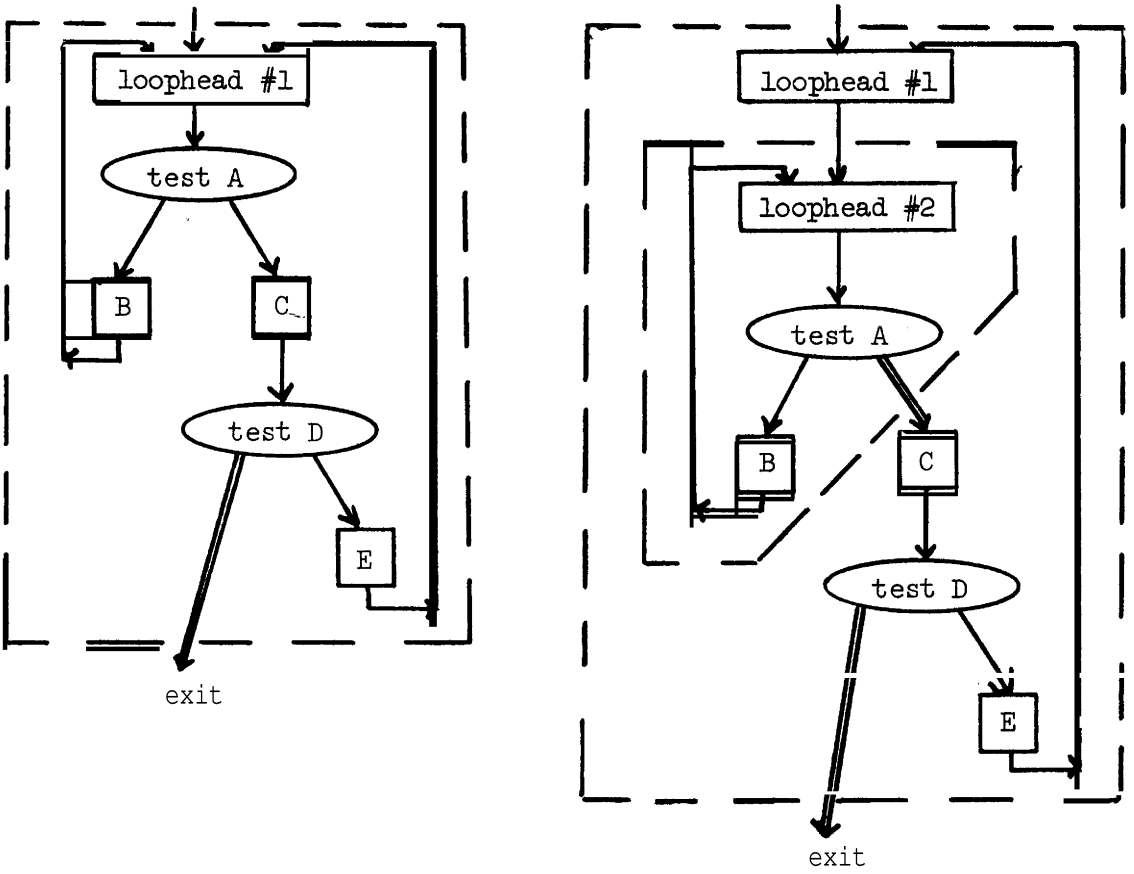


Figure 4 continued.

Third, in order to separate information related to the issue of loop termination, we in general need to modify each loop so that every path around the loop goes through a test which can exit the loop. We make a separate, contained, loop out of any paths which do not exit directly, as in Figure 5. There are no paths around the loops in Rivest's program which require this modification.



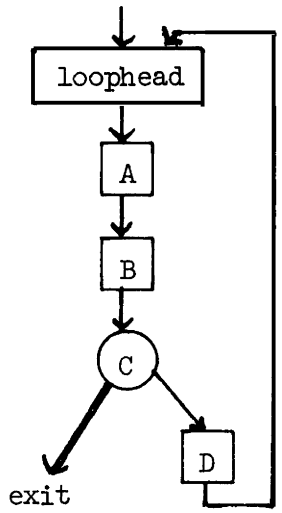
5a. The paths of this flow graph are are described by the regular expression $(A(B+CDE))^*ACD$

5b. The paths of this flow graph are described the regular expression $((AB)^*ACDE)^*(AB)^*ACD$

Figure 5. Example of forcing each path around a loop to go through an exit test.

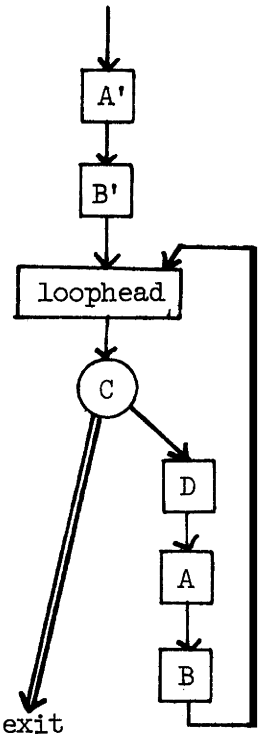
In generating loop termination assertions, we will essentially be stating that "for some k , an exit path is taken on the k-th iteration of the loop". It is convenient for the generation of such assertions to have loops in which the exit tests are near the top of the loop. Specifically, if there are embedded loops, function calls, or complicated calculations between the loophead node and the various tests which exit the loop, then it will be hard to describe the values of the program variables at the test node in terms of their values at the loophead node. Therefore, our fourth modification is to attempt to permute the nodes in the loop so that all exit tests occur immediately after the loophead node. Our modified flow graph for select now looks like Figure 7, with copies of loops 3 and 4, and loop 2 permuted.

This modification cannot always be done, as the third example in Figure 6 shows. However, all loops with exactly one immediate exit arc can be successfully permuted. The effect of these last two modifications is put the programs in nested while format. For another method to accomplish this transformation, see [Ashcroft and Manna].



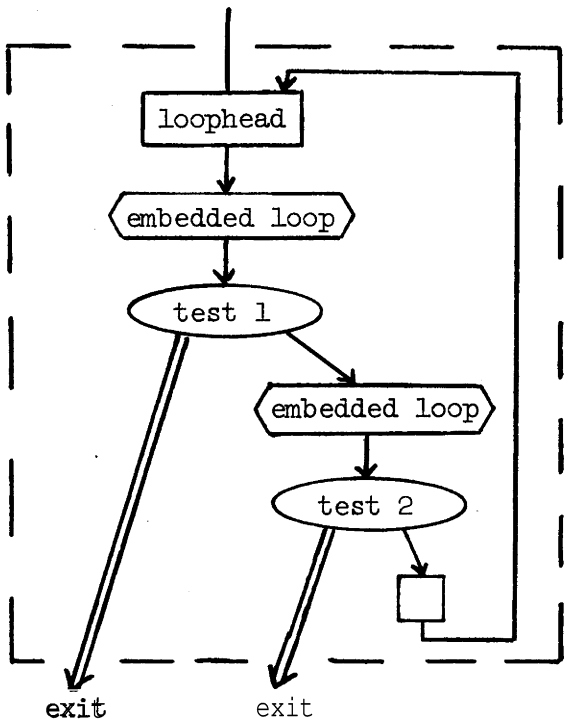
regular expression = $(ABCD)^{*}$ AX

EXAMPLE 6a. Transform this to 6b.



regular expression = AB(CDAB)*C

EXAMPLE 6b. Leading test form of 6a.



EXAMPLE 6c. Cannot move both tests to top of outer loop.

Figure 6. Examples of permuting loops' to create leading tests.

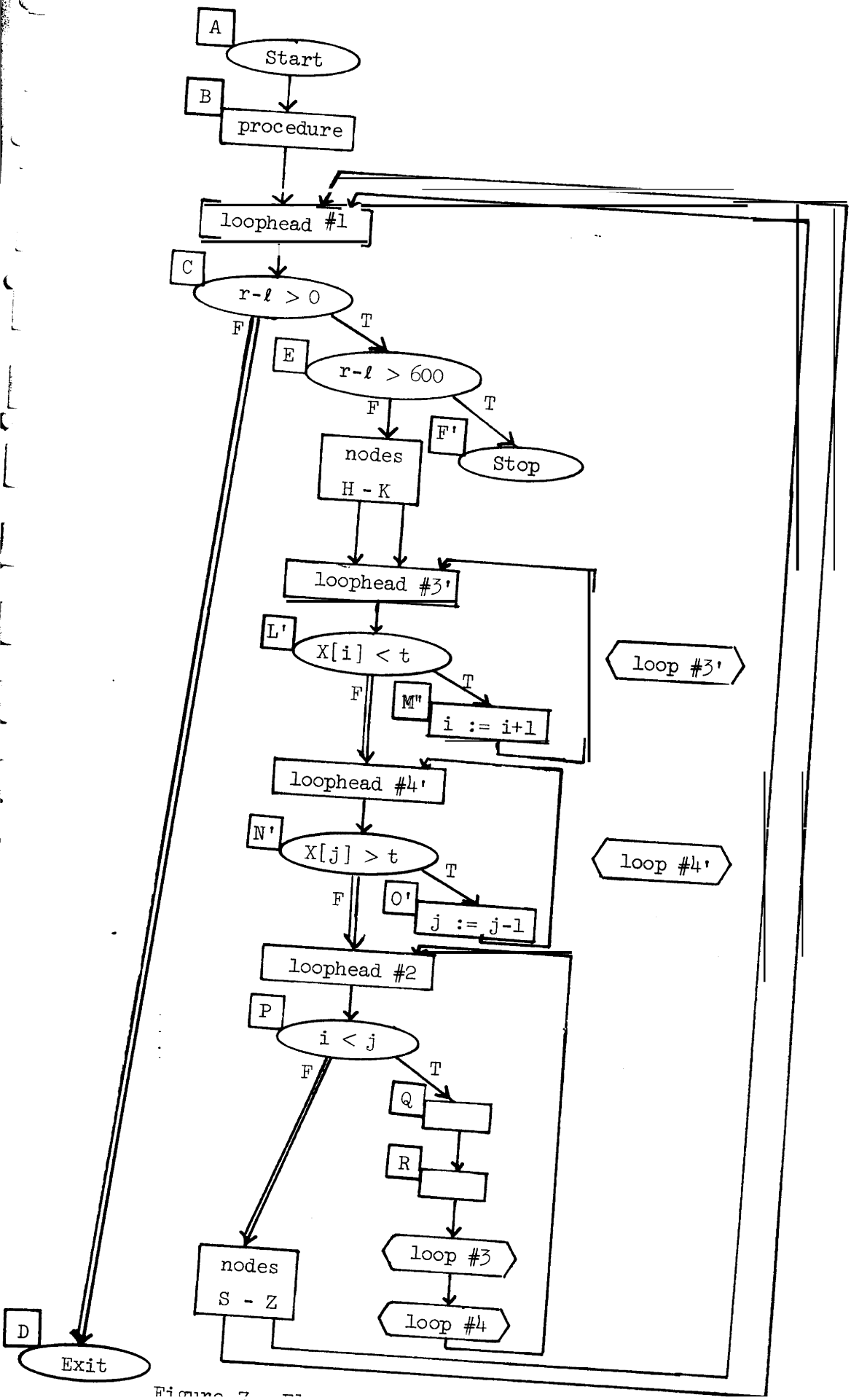


Figure 7. Flow graph of select after forcing into nested WHILE format, with 1-permuted.

With the flow graph put in the desired form, we are now ready for the work of creating and proving assertions about clean termination.

II. Semantic Error Assertions

We start by attaching assertions to the arcs of the flow graph stating that all operations in the subsequent node are well-defined and that no semantic errors are generated. This is essentially an operation-driven process of inserting assertions on all incoming arcs of a node, specifying exactly what conditions must be true for the contained operations to execute cleanly. Typical assertions are:

Operation	Assertion generated
i+j	$i \neq \omega \text{ A } j \neq \omega \text{ A } I_{min} \leq i+j \text{ A } i+j \leq I_{max}$, where ω stands for "undefined", I_{min} is the smallest representable integer in the machine on which the program will execute, and I_{max} is the largest representable integer.
A[i]	$i \neq \omega \wedge A_{\ell} \leq i \text{ A } i \leq A_u$, where A_{ℓ} is the lower bound for legal subscripts of A and A_u is the upper bound.
i := j	$j \neq \omega$.

See Figure 8 for an example of this assertion synthesis process. Note that if a node has many incoming arcs, the same set of assertions will be attached to each arc. A detailed example of assertion generation appears in [Sites].

If all the assertions generated at this stage are proved to be true, then the program contains no semantic errors, i.e., it does not "blow up" during execution, perhaps with a run-time error message.

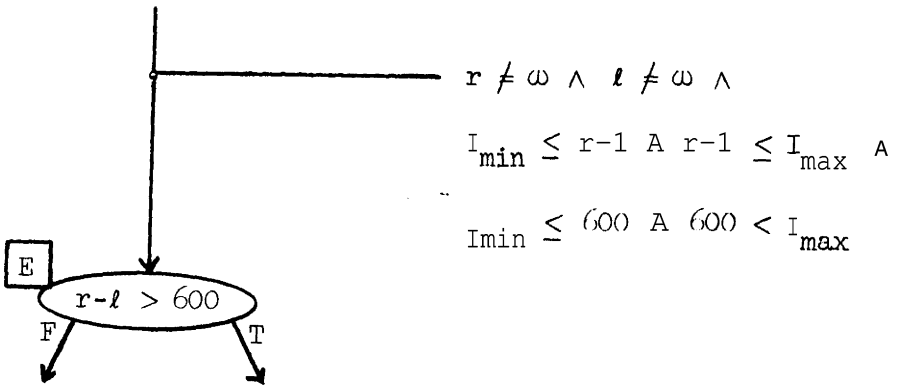


Figure 8. Example of assertion generation.

In most cases, the generation is quite simple, but some complications arise in handling procedure calls:

(i) Arguments passed to value parameters of a procedure are treated like the right-hand side of an assignment -- the argument expression must be defined and must not generate semantic errors when it is evaluated at the point of call.

(ii) Procedures with name parameters must be handled strictly according to the copy rule, making a unique copy of the procedure for each call and logically inserting the body of the procedure instead of that call. This is the only way to properly reflect the side effects which can result from tricky use of name parameters. It is also a reason that Algol 60 recursion is hard to analyze mechanically.

(iii) Procedures with array arguments have the problem that the procedure does not specify the legal lower and upper bounds for the subscripts. Either of two strategies can be adopted for generating and proving assertions about-subscripts in the proper range: symbolic

names (like A_ℓ and A_u) can be used in all the assertions, and the proof techniques can push back to the entry point of the procedure any assertions (restrictions) which must be true on entry in order to avoid subscript range errors; alternately, the programmer can supply an extra statement to the proof system, describing the bounds for each array. The first strategy is equivalent to asking, What are the necessary array bound conditions for this procedure to always terminate cleanly?" The second strategy is equivalent to saying, "Here are the conditions which will always be true when the procedure is called; are they sufficient to guarantee clean termination?"

If the programmer has definite assumptions about ranges of array bounds in his mind, then it is best to state them to the proof system. Failure to do so forces the system to try to synthesize the equivalent information, a process which may well fail.

In the assertions and proofs which follow, it will be assumed that the proof system has been told that $X[\ell_0:r_0]$ is the declaration for array X , where ℓ_0 equals the value of ℓ upon entry to select , and r_0 equals the value of r upon entry. This binding allows the subscript range assertions to be independent of the fact that the variables ℓ and r change value as the program executes.

III. Loop Termination Assertions

Loop termination assertions are harder to generate than semantic error assertions because the goal is much more abstract. For semantic errors, the assertions generated are a straightforward function of the language definition and compiler/computer implementation restrictions. For loop termination, however, synthesizing the proper assertion may well be harder than proving it true.

Loop termination can be approached on a wide variety of levels of abstraction. One extreme is to assert that control passes through each loophead node a finite number of times. However, such a statement doesn't lend itself to direct proof. Another extreme is to require all loops to be FOR loops or DO loops in which the step and limit are evaluated exactly once and the iteration variable cannot be changed inside the loop. Such loops terminate by definition (if a zero step is prevented). In between these extremes are some useful strategies.

One strategy is to use the taking of an exit branch as a goal to drive the assertion generation. For any loop, asserting that it terminates is equivalent to asserting that $\exists k \geq 1$ such that on the k-th iteration of the loop, one of the sets of tests leading to an exit arc will be true. Given the form of loops with leading tests that we have specified, it is easy to generate such an assertion mechanically.

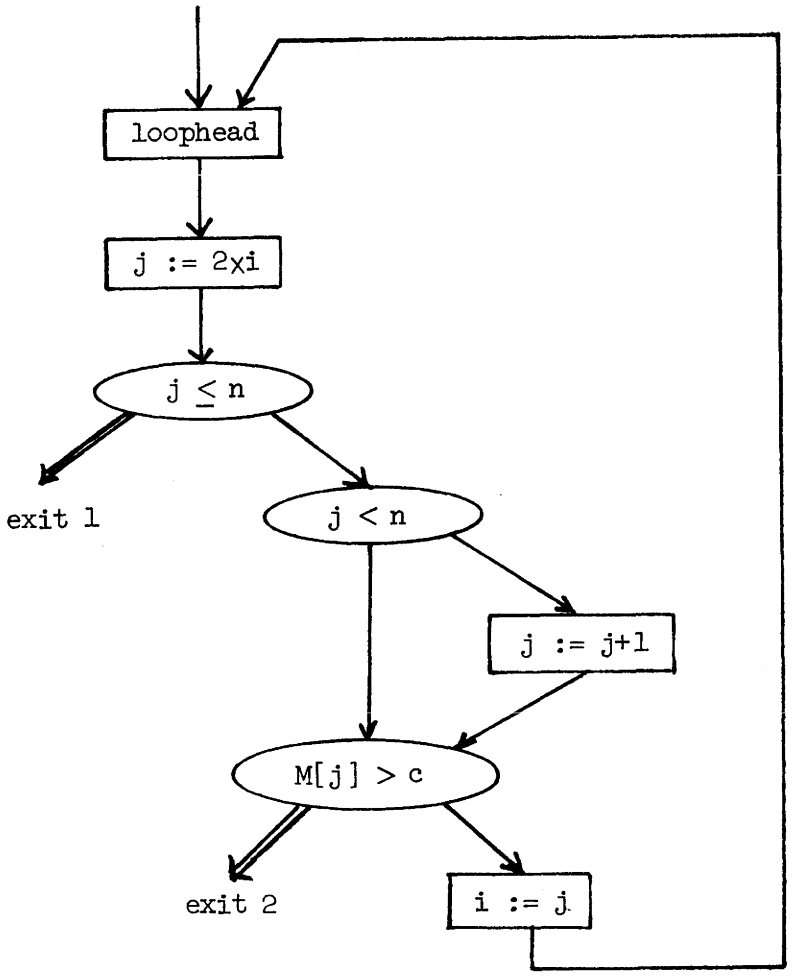


Figure p. Example flow graph for discussion of loop termination assertions.

Using the notation i_k to mean the value of variable i at the loophead node on the k-th iteration of the loop, the termination assertion for Figure 9 is:

$$\exists k \geq 1 \text{ s.t. } (2 \times i_k > n_k) \vee$$
$$(2 \times i_k \leq n_k \wedge 2 \times i_k \geq n_k \wedge M_k[2 \times i_k] \leq c_k) \vee$$
$$(2 \times i_k \leq n_k \wedge 2 \times i_k < n_k \wedge M_k[2 \times i_k + 1] \leq c_k)$$

This expression was derived by substituting for j its value in terms of the values of the program variables at the top of the loop. Note that depending upon the path taken, this value is either $2 \times i_k$ or $2 \times i_k + 1$.

While an assertion such as the one above can be generated from any loop described in Section II, it is in general an unsolvable problem to prove that the assertion is true. However, a small variety of techniques based on monotonic variables, finite sets, and search loops can prove the termination of most loops encountered in practical programs.

Also, this strategy of generating a $\exists k \ldots$ assertion sometimes allows a proof system to state that a loop definitely never terminates. For instance, if the statement i := j were accidently left out of the example loop above, then it can be shown that all variables in the assertion are invariant within the loop. Thus, the existential quantifier can be dropped, and the remaining assertion states that the program exits the loop on the first iteration. If this assertion is true, the loop terminates immediately; if it is false, the loop never terminates.

Another strategy is to use the existence of a monotonic variable which does not overflow as a goal to drive the assertion generation. If

a loop contains a monotonically increasing or decreasing variable which never overflows when it is updated, then the variable takes on a finite number of values, so the loop terminates. Assertions specifying no overflow are already generated by the semantic error assertion mechanism, so if a monotonic variable is found inside a loop, no other assertions are needed: if the existing "no overflow" assertions are true, then the loop terminates.

This simple strategy is beautiful when it works, but of course it won't always work. For example, if the assertions are not true and an overflow may occur, the proof system may not be able to state directly that the program has an infinite loop. Also, if no monotonic variables are found, this strategy doesn't suggest anything else to try.

Using the first strategy, we generate the following loop termination assertions for Rivest's modified program:

Loop #1 $3k \geq 1$ s.t. $(r_k - \ell_k < 0)$ v $(r_k - \ell_k > 600)$

Loop #3' $3k \geq 1$ s.t. $X_k[i_k] \geq t_k$

Loop #4' $3k \geq 1$ s.t. $X_k[j_k] \leq t_k$

Loop #2 $3k > 1$ s.t. $i_k \geq j_k$

Loop #3 $3k \geq 1$ s.t. $X_k[i_k] \geq t_k$

Loop #4 $3k \geq 1$ s.t. $X_k[j_k] \leq t_k$.

Using the second strategy, we find that in loops 3', 4', 2, 3, and 4 there are monotonic variables. We are going to be in trouble later, however trying to prove that, at node M , i := i+1 does not overflow. In loop 1, we have another problem. Neither ℓ nor r are

monotonic, but their difference r-1 is strictly decreasing. Using the second strategy, it is not clear how to discover that r-1 is a relevant expression.

IV. Proofs of Assertions

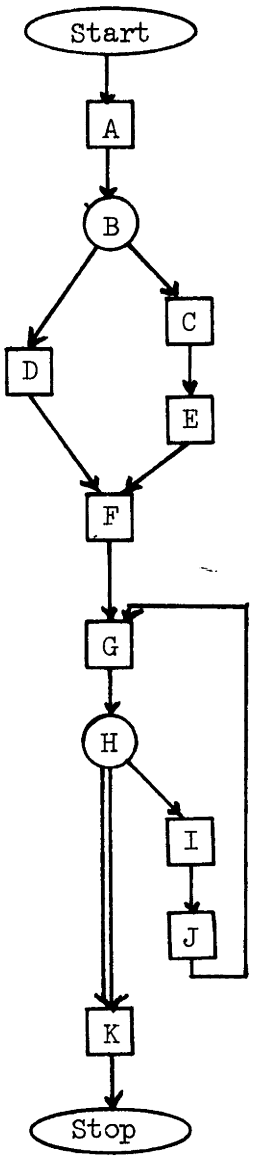
We now order the nodes in the modified flow graph according to the following rules.

(1) Logically reduce each loop in the program to a single node (a loop is the set, of nodes in an interval, plus the **loophead** node, minus all nodes in the interval which have no path leading back to the **loophead** node).

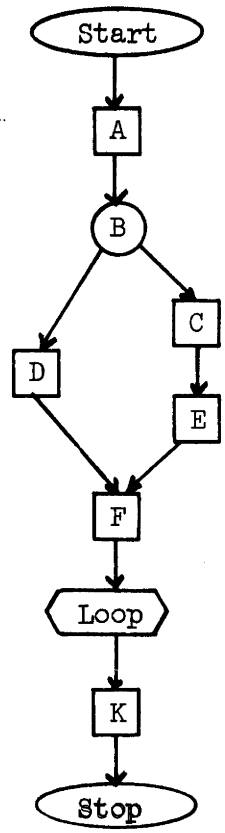
(2) Topologically sort the nodes in the reduced graph, using the (directed) arcs as the ordering.

(3) For each node in the reduced graph which represents a loop, topologically sort the nodes within the loop, ignoring all latchback arcs, then insert those nodes in the main topological ordering as a single group, so that all nodes in the loop precede any nodes which followed the loop in the reduced ordering.

(4) Apply step 3 until all loops have been expanded (see Figure 10).



(1) reduced graph



(2) Ordering: ABDCEF (loop) K

(3) Ordering within (loop)

G H I J

(4) Final ordering

ABDCEF GHIJ K

Figure 10. Example of node ordering.

The proof mechanism will visit the nodes of the graph in the order specified above. At each node, it will attempt to prove all assertions on all incoming arcs. The order specified has a few fairly obvious properties: (1) Except for loophead nodes, whenever a node is visited by the proof mechanism, all predecessor nodes will have been visited. (2) For loophead nodes, all initial entry predecessor nodes will have been visited. (3) If a predecessor node is inside a loop, all nodes in that loop will have been visited. (4) The question "Is node X inside loop N ?" can be answered with a simple range test.

If a program consists of many non-recursive procedures, we process the innermost Procedure first, so that each procedure will have been completely processed before any calls to it are encountered.

We will process nodes in the modified flow graph of select (Figures 7 and 1) in the order:

AB (loop 1) D

or AB CEF'HIJK L'M' N'O' PQR LM NO STUVWXYZ D

The processing at each node visited will be discussed in terms of the model in Figure 11.

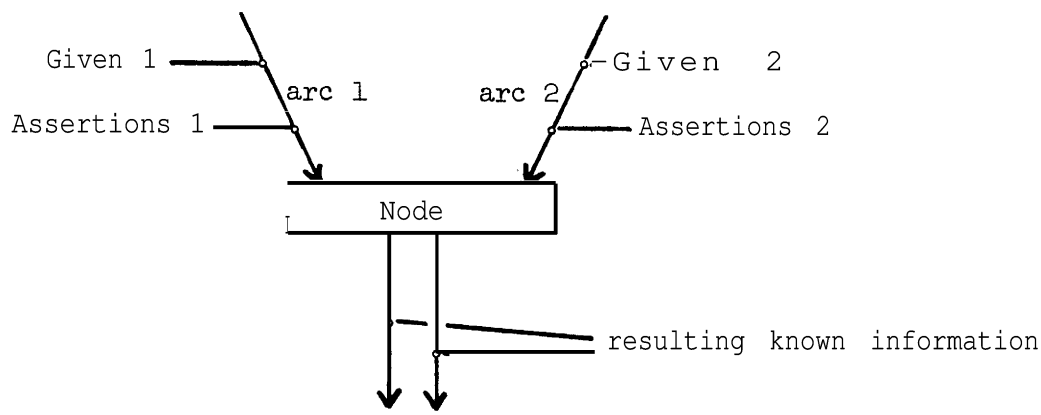


Figure 11. Model node for assertion proofs.

If the node being visited is not a loophead node, we try to prove the assertions on each incoming arc, using the "given" information attached to the same arc (the arc leaving the START node has empty "given" information attached to it). More precisely, we try to prove the theorems

Given 1 ⊃ Assertions 1

Given 2 ⊃ Assertions 2 .

If the node being visited is a loophead, we try to prove the assertions on just the initial entry (non-latchback) arc(s).

If any assertion cannot be proved conclusively true, then a message is printed for the--user.

To catch mistakes or state restrictions as early in the program as possible, we try to move assertions which cannot be proven true back toward the entry point of the procedure. This is purely an optional step, in which we try to help the user by moving unproved assertions to the earliest place in his program that he is likely to want to insert a fix for the bug. We do this movement by taking an assertion and attempting to "pass it through" the preceding node, attaching the (possibly modified) assertion to each incoming arc, as shown in Figure 12.

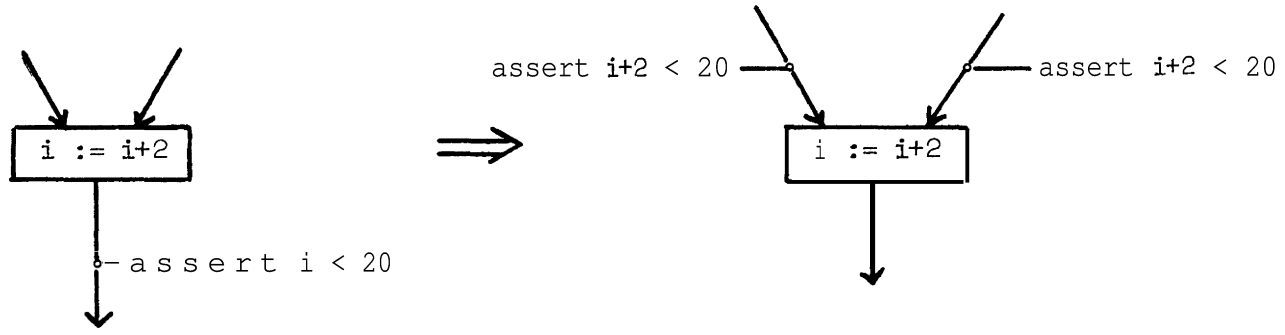


Figure 12. Passing an unproved assertion back through a node.

Sometimes, it will not be possible to move an assertion because the operations inside the node are not reversible (read, or perhaps a procedure call). When pushing an unproved assertion through a loophead node, we don't attach it to any latchback arcs, since then the assertion would be pushed around the inside of the loop forever. In general, it is not useful to push an assertion back through a loop which modifies any of the variables in the assertion, as in Figure 13.

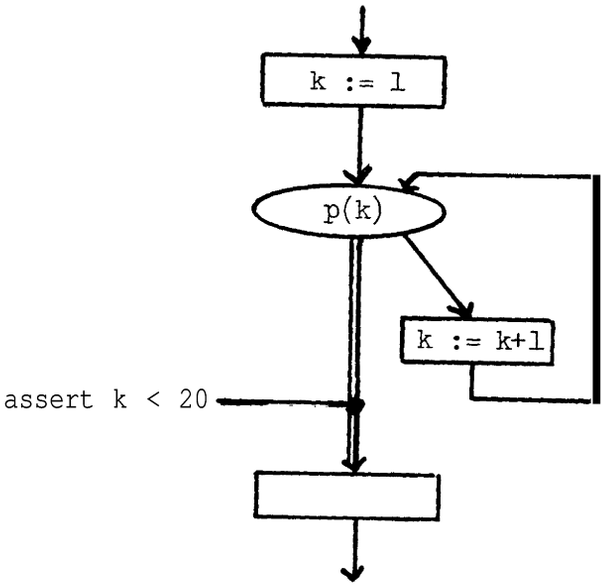


Figure 13. The unproved assertion cannot be pushed back usefully.

If we are visiting a loophead node, the processing is more complicated. Our model node now looks like the one in Figure 14.

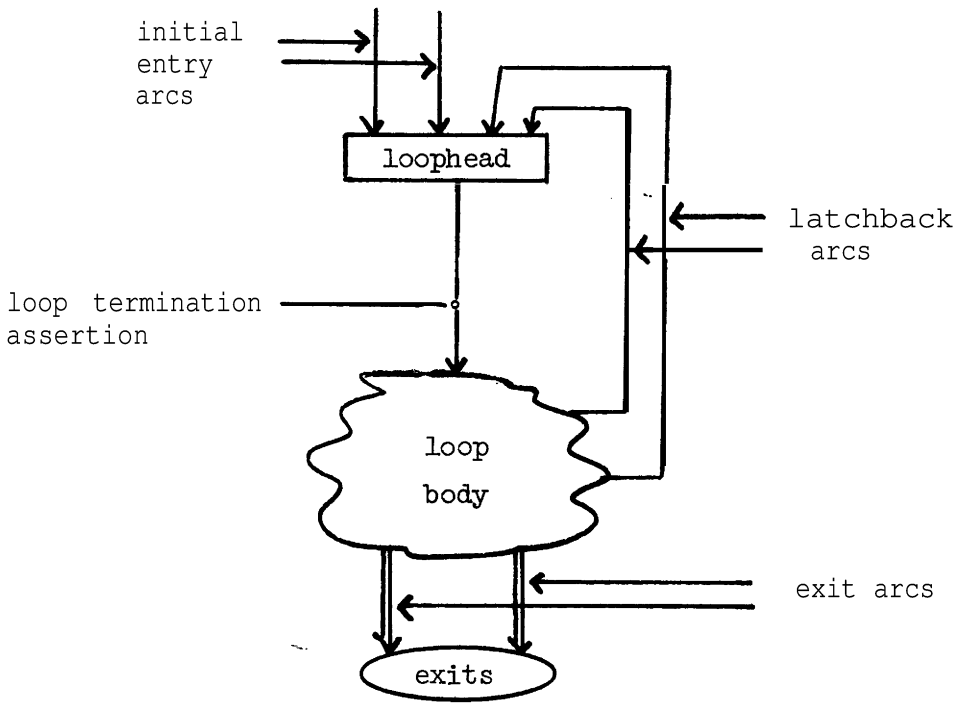


Figure 14. Model node for **loophead** processing.

First, we process all assertions on the initial entry arcs as described above. Then we set all program variables to dummy symbolic values, say $i = i_0$, $j = j_0$, $x = x_0$, and visit all the nodes in the loop body, propagating and merging the "given" information based on these symbolic values, but not proving any assertions. We do not follow the exit arcs, and we stop when the given information has been established for all latchback arcs. (The given information on a latchback arc might be something like $i = i_0+1 \text{ A } i \leq 11$.)

We then feed the initial entry conditions and the once-around-the-loop symbolic expressions on the latchback arcs to an induction routine that tries to synthesize a range or set of values which each program variable takes on at the **loophead** node. In this induction routine, particular

care should be given to detecting variables which are invariant within the loop, and those which are monotonic. We attach the induced ranges and relationships as "given" information on the arc leaving the loophead node. Using this information, we take a second pass over the body of the loop, processing it in the normal way. Note that a loop nested n levels deep will be processed a total of 2^n times.

After visiting a node and handling the assertions on its incoming arcs, our next step is to create the resulting "given" information to attach to the exit arc(s). This information consists of all input arc given information, plus all input assertions (since if we leave the node cleanly, all the input assertions must have been true), modified by any assignments which occur inside the node. If the node being visited is a test, we add the relation or its negation to the true and false exit arcs respectively.

If two or more incoming arcs specify different "given" conditions, we take the most encompassing information, i.e., if one incoming arc specifies $i \geq 10$, and another $i \geq 11$, we use $i \geq 10$. Whenever information is lost by this merging process, we mark the resulting item (the examples below use an asterisk) so that if later it becomes significant whether i is exactly equal to 10, we will know that there is a possibly useful refinement to the information $i \geq 10^*$.

V. Application of Proof Process to Rivest's Program

Referring back to Figure 7, we find that the modified flow graph for select consists of loops nested three deep, as shown in Figure 15.

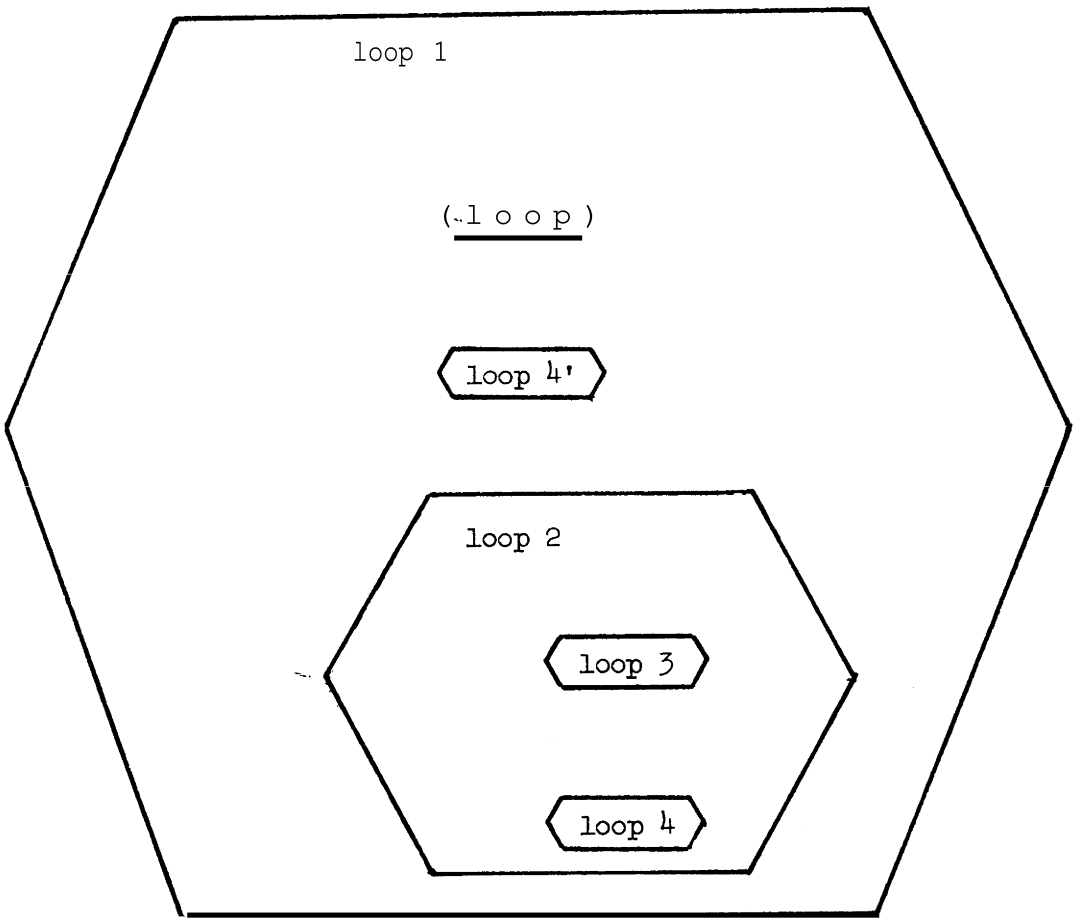


Figure 15. Structure of loop nesting in select.

We begin the first pass over loop 1 by attaching the symbolic values

$$r = r_0 \ , \ \ell = \ell_0 \ , \ k = k_0 \ , \ i = i_0 \ , \ j = j_0 \ , \ t = t_0$$

$$\text{and} \quad x = X_0$$

to the arc between the loophead #1 node and node C . On this first pass, we will analyze the nodes inside loop 1 and develop induction expressions relating the values of all variables after one iteration through the loop to their values at the beginning of that iteration; the symbols

$r_0, \ell_0, \ldots, X_0$ represent these initial values. Following the process given in Section IV, we visit (in order) nodes C , E, F' , and H , developing the following information on the arc entering node I (refer back to Figure 1 for the exact content of the nodes involved):

$$r = r_0 \wedge \ell = \ell_0 \wedge \quad k = k_0 \wedge t = t_0 \wedge \quad X = X_0 \wedge$$

$$i = \ell_0 + 1 \text{ A}$$

$$j = r_0 - 1 \text{ A}$$

$$0 < r_0 - \ell_0 \leq 600$$

One of the biases in our processing should be to reflect relationships in terms of initial values at the top of the loop; thus we write $0 < r_0 - \ell_0$ instead of $0 < r-1$, the latter signifying the values of r and ℓ at the arc to which the relation is attached. If, say, r is changed, the relation $0 < r - \ell$ may no longer be true, while $0 < r_0 - \ell_0$ still would be.

After processing node I , we attach the following information to the arc leading to node J :

$$r = r_0 \wedge \ell = \ell_0 \wedge \quad k = k_0 \wedge$$

$$i = \ell_0 + 1 \wedge$$

$$j = r_0 - 1 \wedge$$

$$0 < r_0 - \ell_0 \leq 600 \wedge$$

$$X[k_0] = X_0[\ell_0] \wedge$$

$$X[\ell_0] = X_0[k_0] = t$$

$$X[m] = X_0[m] \quad \text{for } m \neq k_0, \ell_0 \ .$$

To develop the relations about X , we needed to examine two sub-cases in node I : (i) $k_0 \neq \ell_0$, and (ii) $k_0 = \ell_0$. In order to keep the size of the information attached to the arcs manageable (and hence to

keep the complexity of the proofs using this information manageable), our information development algorithms must be biased toward mimicking the human trait of finding useful lemmas which are true for as many different cases as possible. In analyzing the assignments in node I , we should notice that the degenerate case $k_0 = \ell_0$ does not upset any of the relationships from the general case: the relations above are a true and complete description of the effect of executing node I , even when we substitute " k_0 " for " ℓ_0 " in all relations, so after this checking we find no need to distinguish the degenerate case.*/

In processing nodes J and K , we have to examine the degenerate cases $r_0 = k_0$ and $\ell_0 = k_0$ (r_0 cannot equal ℓ_0 because $0 < r_0 - \ell_0$; if we fail to use this fact on the first pass through loop 1, it gets much harder to keep track of all the assignments to elements of X).

On the true exit from node J , we attach the lemma:

$$X[r_0] < t \wedge t = X[\ell_0]$$

(which is true independent of the relationships between ℓ_0 , r_0 and k_0), and the more detailed information

$$(r_0 \neq k_0 \wedge X_0[r_0] < X_0[k_0]) \vee (r_0 = k_0 \wedge X_0[\ell_0] < X_0[k_0]) \ .$$

After the exchange in node J , the lemma becomes:

$$X[\ell_0] < t \ A \ t = X[r_0]$$

On the false exit from node J , we attach the lemma

$$t = X[\ell_0] \ A \ t \leq X[r_0] \ .$$

---

*/ If the third assignment had been $X[\ell] := t+1$, we would have the following relationships, which do distinguish the two cases:

$$(k_0 \neq \ell_0 \wedge X[k_0] = X_0[\ell_0] \wedge t = X_0[k_0] \wedge X[\ell_0] = X_0[k_0]+1)$$

$$\vee \ (k_0 = \ell_0 \wedge t = X_0[k_0] \wedge X[k_0] = X_0[k_0]+1) \ .$$

In the second case, $X[k_0] \neq X_0[\ell_0]$.

Moving on to the loophead #3' node in Figure 7, we have to merge the information on the two initial entry arcs. In particular, we merge the two relationships

$$X[\ell_0] < t \wedge t = X[r_0]$$

$$X[r_0] \geq t \wedge t = X[\ell_0]$$

into

$$X[\ell_0] \leq t \leq X[r_0]^*$$

where the asterisk indicates that a refinement of the information is available by considering the incoming paths seaprately. This crucial fact will allow us to prove that the while loops on i and j (3', 4', 3, and 4) all terminate, and do so without producing a subscript-out-of-range error.

The remainder of the processing will only be sketched.

In loop 3', i takes on the set of values $\{\ell_0+1, \ell_0+2, \ell_0+3,$ ● ⊠⊠❁ 🔊 Included in this set is r_0, since $0 < r_0-\ell_0$ implies $\ell_0+1 \leq r_0$. Therefore, the loop termination assertion for loop 3', $3k \geq 1$ s.t. $X_k[i_k] \geq t_k$, is at worst true when i reaches r_0. Thus loop 3' terminates, and included in the information on the exit arc are the relations:

$$\ell_0+1 \leq i \leq r_0 \wedge$$

$$X[i] \geq t .$$

Similarly, loop 4' terminates with the following included in its exit information:

$$\ell_0 \leq j \leq r_0-1 \wedge$$

$$X[j] \leq t \wedge$$

$$\ell_0+1 \leq i \leq r_0 \wedge$$

$$X[i] \geq t \wedge$$

$$X[\ell_0] \leq t \leq X[r_0]^*$$

The first pass through loop 2 establishes that i is strictly increasing and that j is strictly decreasing. This monotonicity means that the test i < j will eventually be false, so loop 2 terminates if loops 3 and 4 do. The second pass through loop 2 combines this information with the exit conditions from loop 4' and with the truth of the test i < j to establish that the exchange in node Q never changes $X[\ell_0]$ or $X[r_0]$. Thus, the relation $X[\ell_0] \leq t \leq X[r_0]$* is invariant in loop 2, so on the second pass through loop 2, loops 3 and 4 can be shown to terminate.

For our present purposes, the only interesting thing about nodes S thru Z is that either or both of the assignments

```
ℓ := j+1
r := j-1
```

are done in the context

$$\ell_0 \leq j \leq r_0 \quad .$$

Thus, the quantity r-1 is strictly smaller than $r_0 - \ell_0$ after one iteration of loop 1, proving the termination condition for loop 1 $3k \geq 1$ s.t. $r_k - \ell_k \leq 0$. This completes the first pass of processing loop 1.

On the second pass through loop 1, the initial values of r and ℓ and the pseudo-declaration for X (discussed in Section II(iii) above) are used to prove all the semantic error assertions in the loop. These assertions deal mostly with subscript range errors, overflow errors, and undefined variable errors. The proofs of all these are quite easy, given the information on the bounded ranges of i , j , r , and ℓ gathered on pass 1.

VI. Conclusion

We have explored a structured collection of techniques for mechanically proving that a program terminates cleanly. We then applied these techniques to the proof of clean termination of a non-trivial program, suppressing most of the detailed processing which would be done by an actual computer implementation of such a system.

References

[Allen a]. Allen, F. E., "Control Flow Analysis," Proceedings of a Symposium on Computer Optimization, SIGPLAN Notices, July 1970.

[Allen b]. Allen, F. E., "A Basis for Program Optimization," IBM Research Report RC 3138, T. J. Watson Research Center, Yorktown Heights, N. Y., November 1970, pp. 3-6.

[Allen and Cocke]. Allen, F. E., and Cocke, J., "Graph-Theoretic Constructs for Program Control Flow Analysis," IBM Research Report RC 3923, T. J. Watson Research Center, Yorktown Heights, N. Y., July 1972, p. 28 f f .

[Ashcroft and Manna]. Ashcroft, E., and Manna, Z., "The Translation of 'Go To' Programs to 'While' Programs," Information Processing 71, North-Holland Publishing Company, 1972, pp. 250-255.

[Cocke]. Cocke, J., "On Certain Graph-Theoretic Properties of Programs," IBM Research Report RC 3391, T. J. Watson Research Center, Yorktown Heights, N. Y., June 1971.

[Cocke and Schwartz]. Cocke, J., and Schwartz, J. T., "Programming Languages and Their Compilers: Preliminary Notes," Courant Institute of Mathematical Sciences, New York University, New York, N. Y., April 1970, pp. 442-461.

[Floyd]. Floyd, R. W., "Assigning Meanings to Programs," Proceedings of a Symposium on Applied Mathematics, American Mathematical Society, Volume 19, 1967, pp. 19-32.

[Good]. Good, D. I., "Toward a Man-Machine System for Proving Program Correctness," (Ph.D. Thesis at University of Wisconsin), Texas University Computation Center TSN-11, Austin, Texas, June 1970.

[King]. King, J. C., "A Program Verifier," (Ph.D. Thesis), Carnegie-Mellon University, Pittsburg, Pennsylvania, September 1969, 255 pp. Clearinghouse # AD699248.

[Rivest and Floyd]. Rivest, R. L., and Floyd, R. W., "Bounds on the Expected Time for Median Computations," Combinatorial Algorithms, ed. by Randall Rustin, Algorithms Press, 1973, pp. 69-76.

[Sites]. Sites, R. L.,' "Certification of Algorithm 245 TREESORT3: Proof of Clean Termination -- A New Kind of Partial Certification," Companion paper in this report.